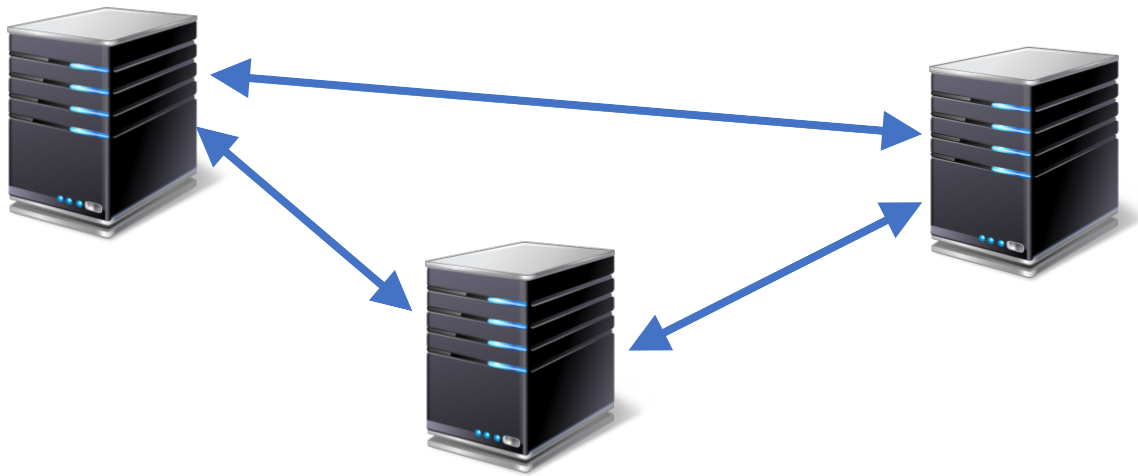
Distributed Systems: Remote Procedure Call (RPC), MapReduce

CS 571: *Operating Systems (Spring 2021)*

Lecture 11

Yue Cheng

What is a distributed system?



- Multiple computers
- Connected by a network
- Doing something together

- A *distributed system* is many cooperating computers that appear to users as a single service

Today's outline

How can processes on different cooperating computers exchange information?

1. Network sockets and raw messages

2. Remote procedure call

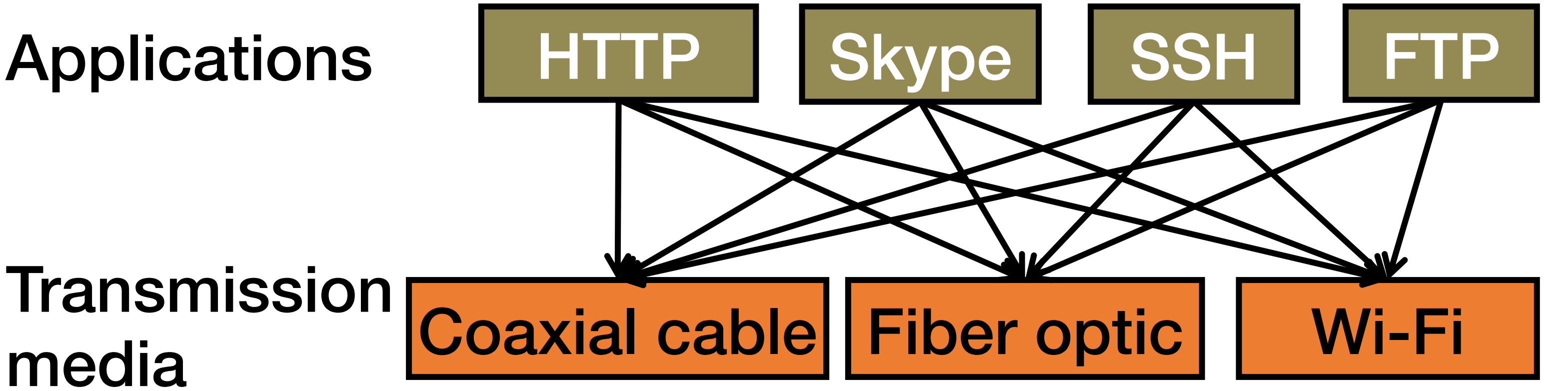
How can large computing jobs be parallelized?

3. MapReduce

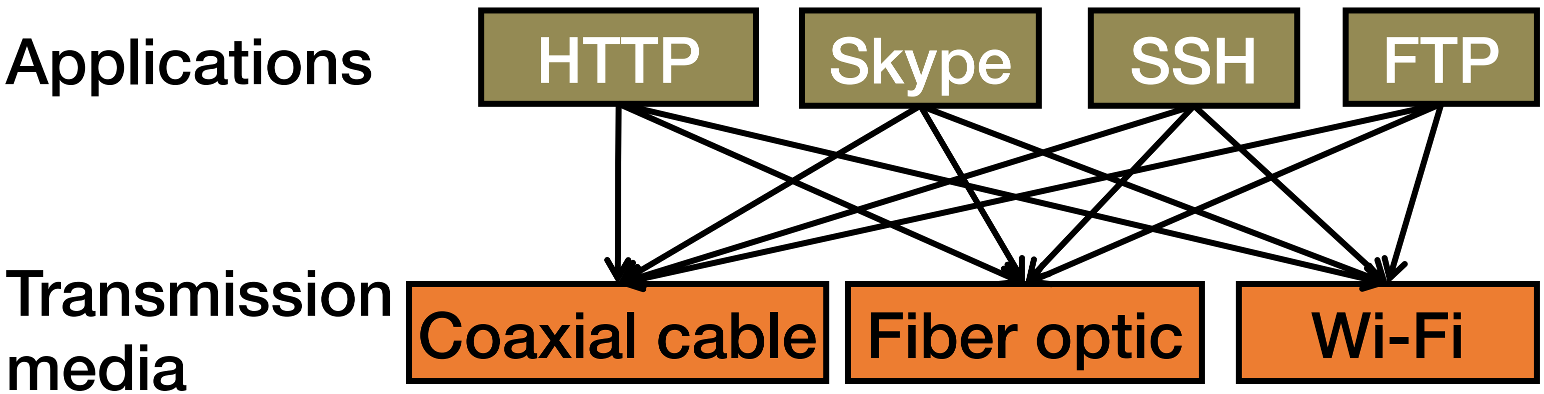
The problem of communication

- Process on Host A wants to talk to process on Host B
  - A and B must agree on the meaning of the bits being sent and received at many different levels, including:
    - How many volts is a 0 bit, a 1 bits?
    - How does receiver know which is the last bit?
    - How many bits long is a number?

The problem of communication

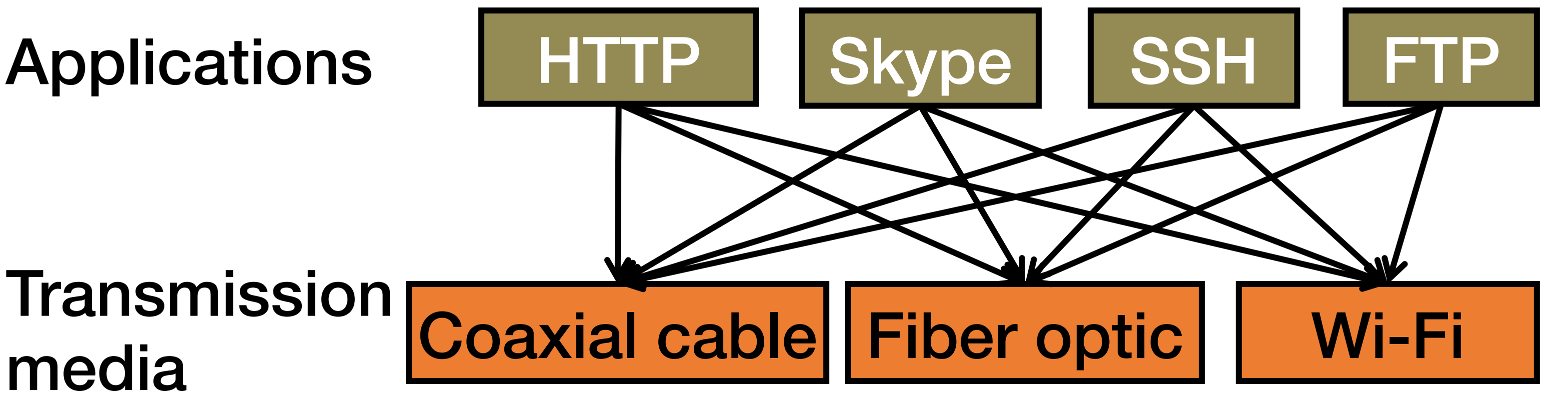


The problem of communication



- Re-implement every application for every new underlying transmission medium?
- Change every application on any change to an underlying transmission medium?

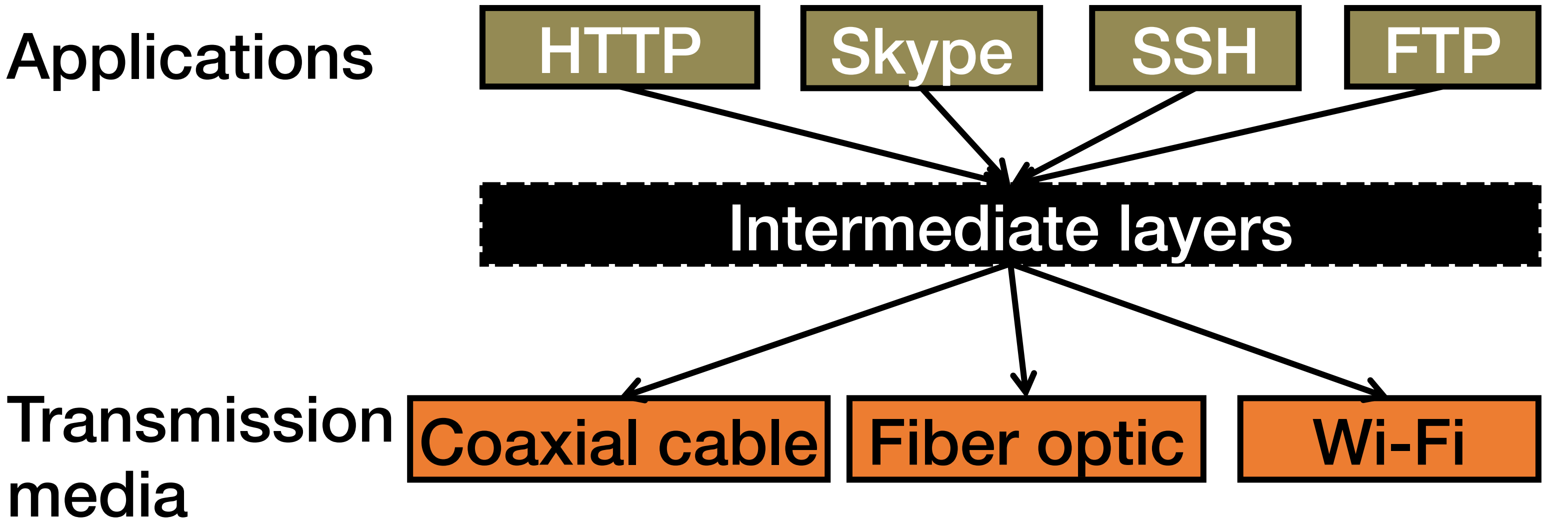
The problem of communication



- Re-implement every application for every new underlying transmission medium?
- Change every application on any change to an underlying transmission medium?

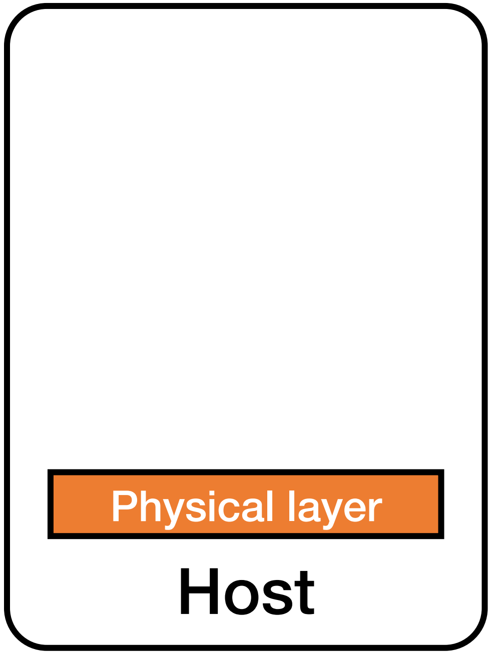
- No! But how does the Internet design avoid this?

Solution: Layering



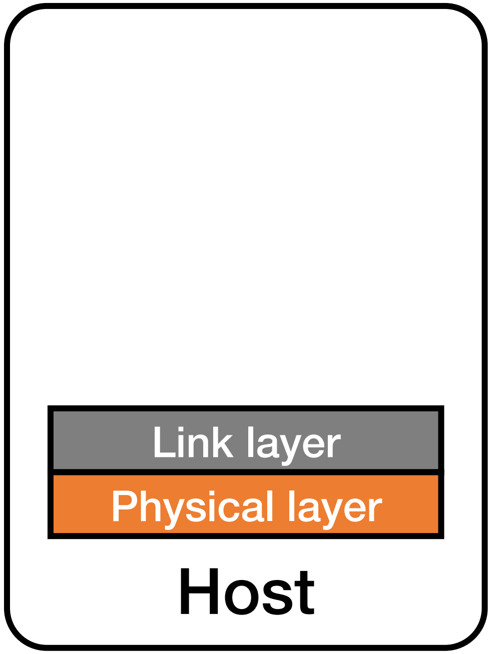
- Intermediate layers provide a set of abstractions for applications and media
- New applications or media need only implement for intermediate layer's interface

Layering in the Internet



- **Physical:** Moves bits between two hosts connected by a physical link

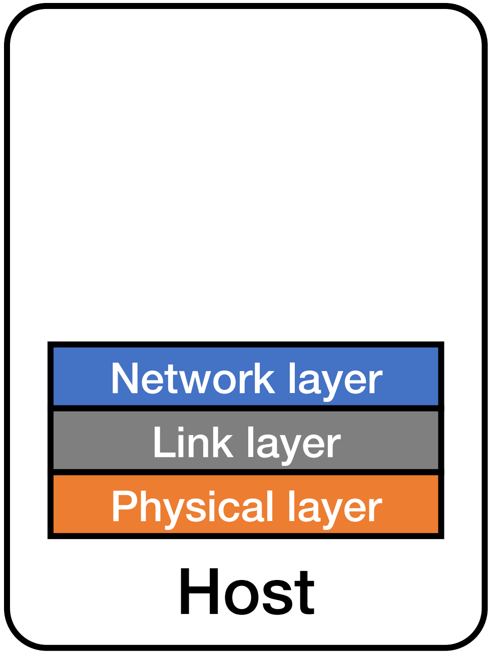
Layering in the Internet



- Link: Enables end hosts to exchange atomic messages with each other

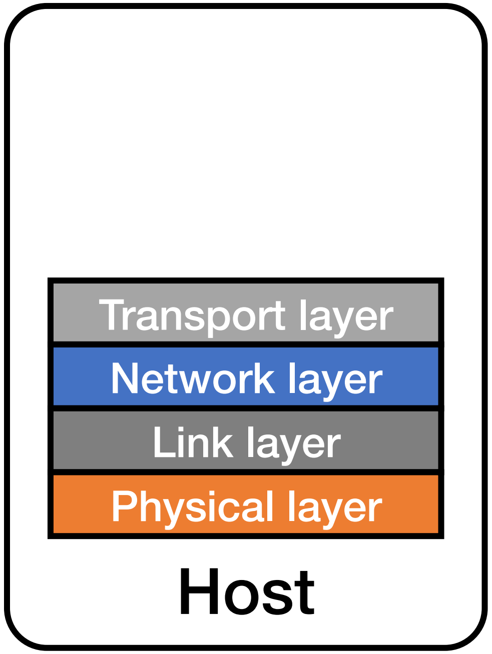
- Physical: Moves bits between two hosts connected by a physical link

Layering in the Internet



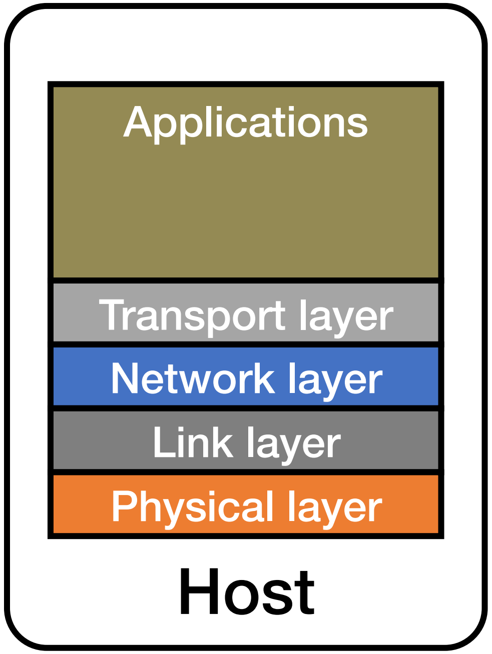
- Network: Deliver packets to destinations on other (heterogeneous) networks
- Link: Enables end hosts to exchange atomic messages with each other
- Physical: Moves bits between two hosts connected by a physical link

Layering in the Internet



- Transport: Provide end-to-end communication between processes on different hosts
- Network: Deliver packets to destinations on other (heterogeneous) networks
- Link: Enables end hosts to exchange atomic messages with each other
- Physical: Moves bits between two hosts connected by a physical link

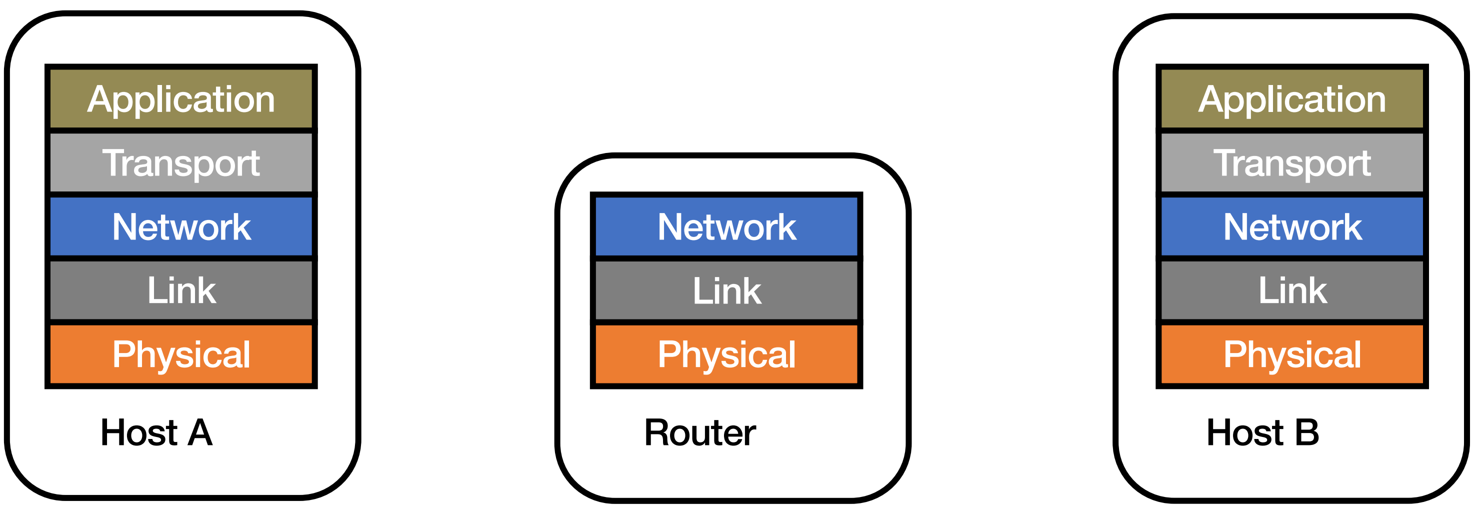
Layering in the Internet



- Transport: Provide end-to-end communication between processes on different hosts
- Network: Deliver packets to destinations on other (heterogeneous) networks
- Link: Enables end hosts to exchange atomic messages with each other
- Physical: Moves bits between two hosts connected by a physical link

Logical communication between layers

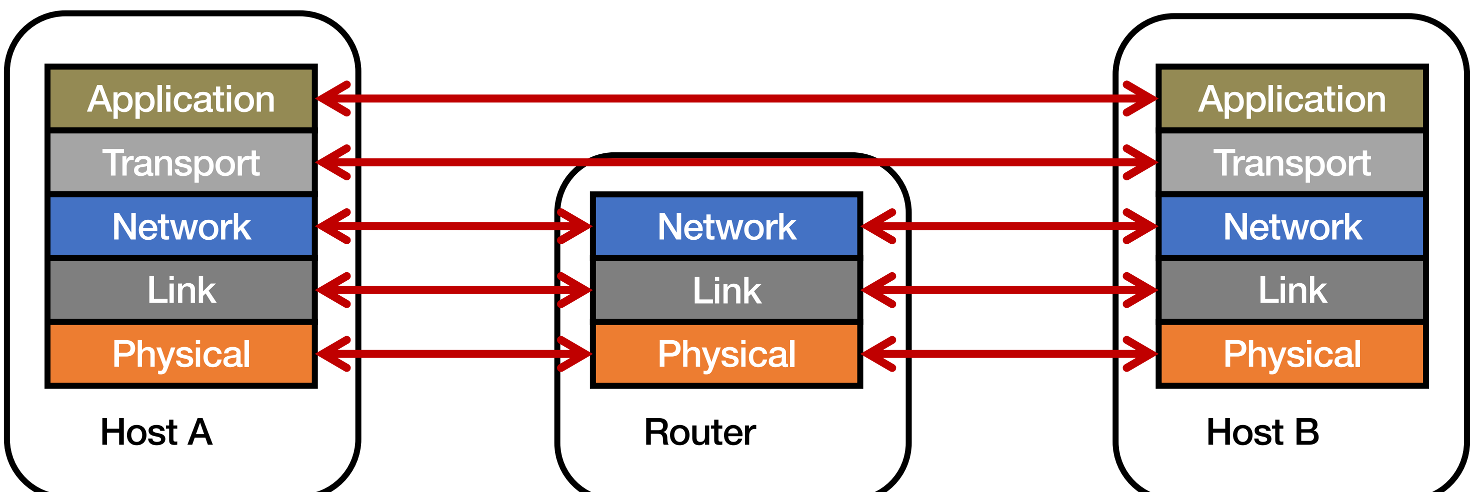
- How to forge agreement on the meaning of the bits exchanged between two hosts?



Logical communication between layers

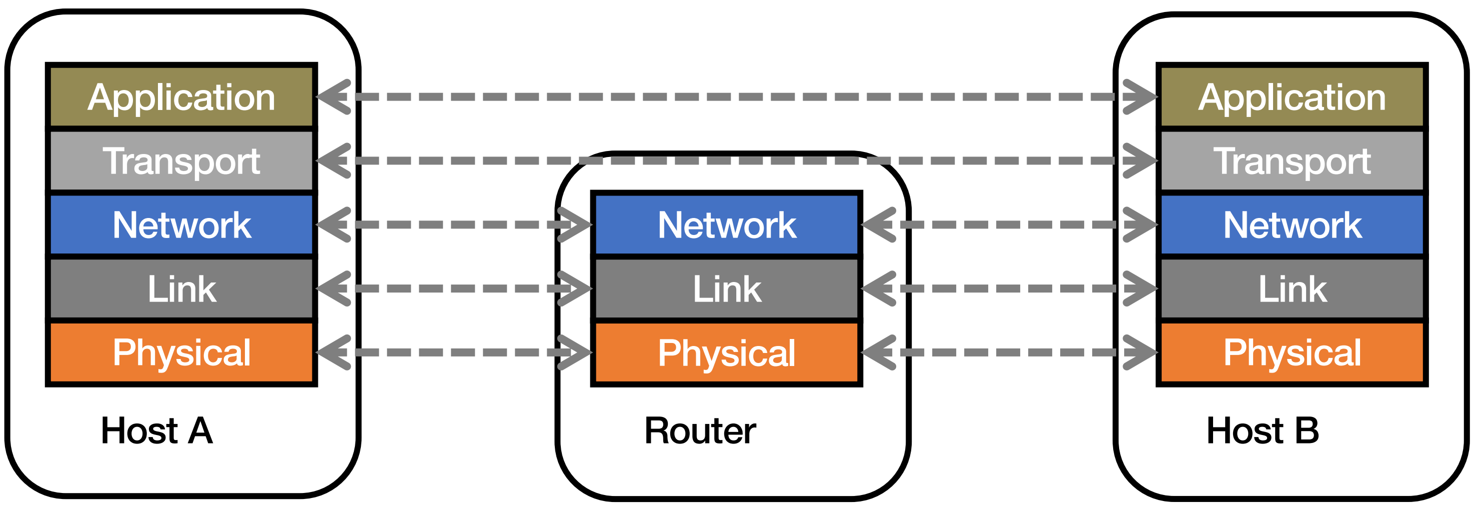
- How to forge agreement on the meaning of the bits exchanged between two hosts?

- **Protocol:** Rules that govern the format, contents, and meaning of messages
  - Each layer on a host interacts with its peer host's corresponding layer via the protocol interface



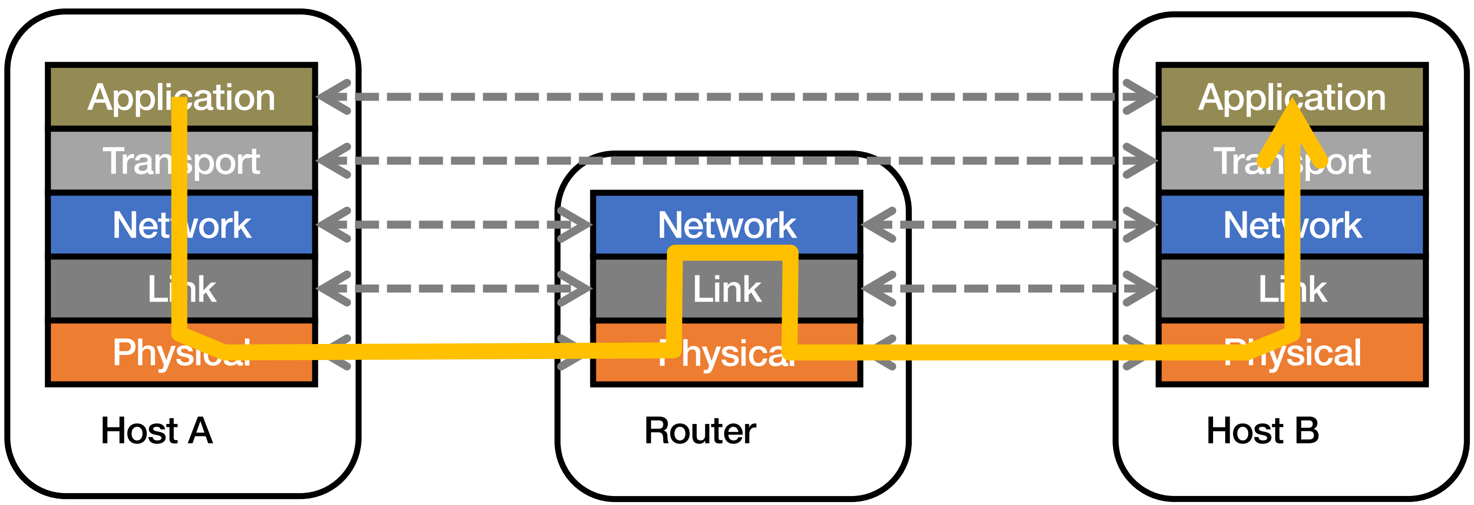
Physical communication

- Communication goes down to the physical network
- Then from network peer to peer
- Then up to the relevant application



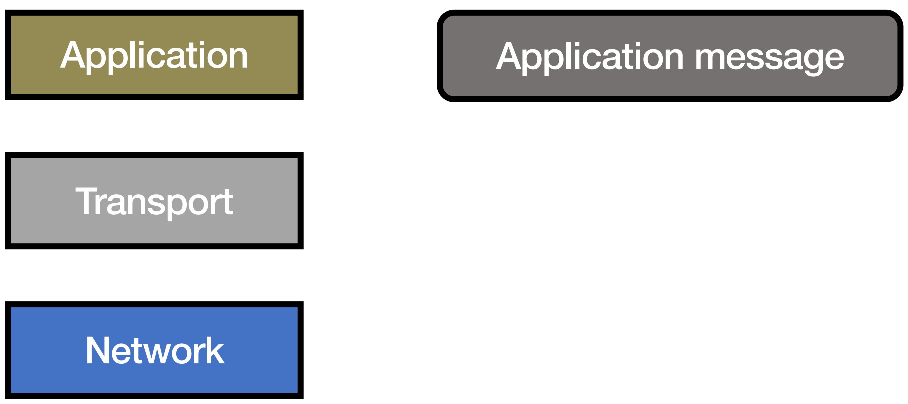
Physical communication

- Communication goes down to the physical network
- Then from network peer to peer
- Then up to the relevant application



Communication between layers

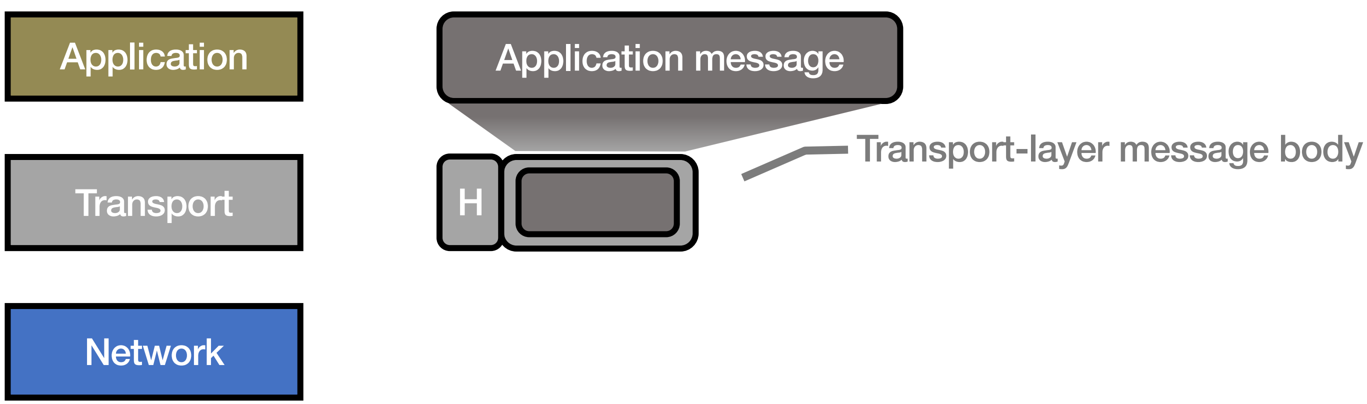
- How do peer protocols coordinate with each other?
- Layer attaches its own header (H) to communicate with peer
  - Higher layers' headers, data encapsulated inside message
    - Lower layers don't generally inspect higher layers' headers



Communication between layers

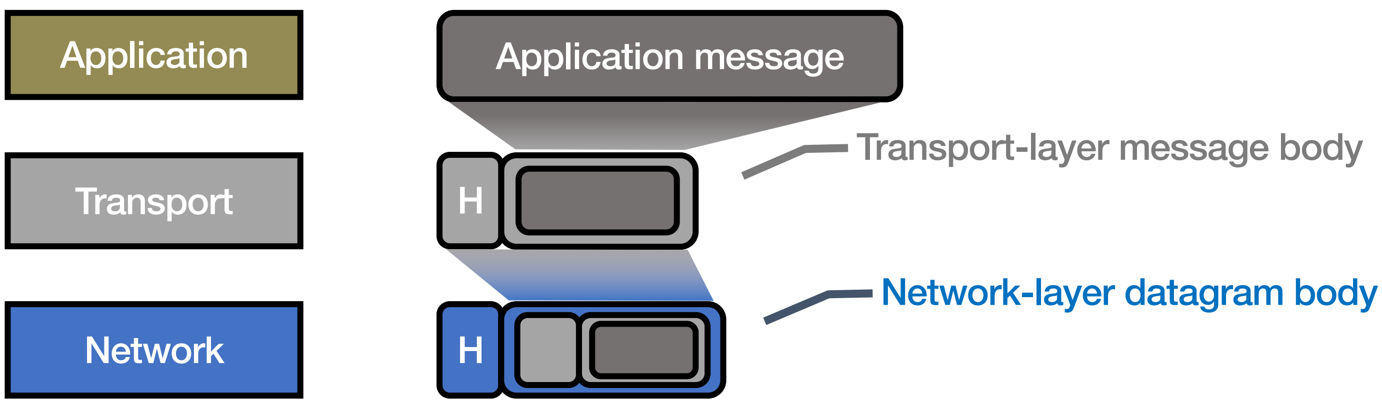
- How do peer protocols coordinate with each other?

- Layer attaches its own header (H) to communicate with peer
  - Higher layers' headers, data encapsulated inside message
    - Lower layers don't generally inspect higher layers' headers



Communication between layers

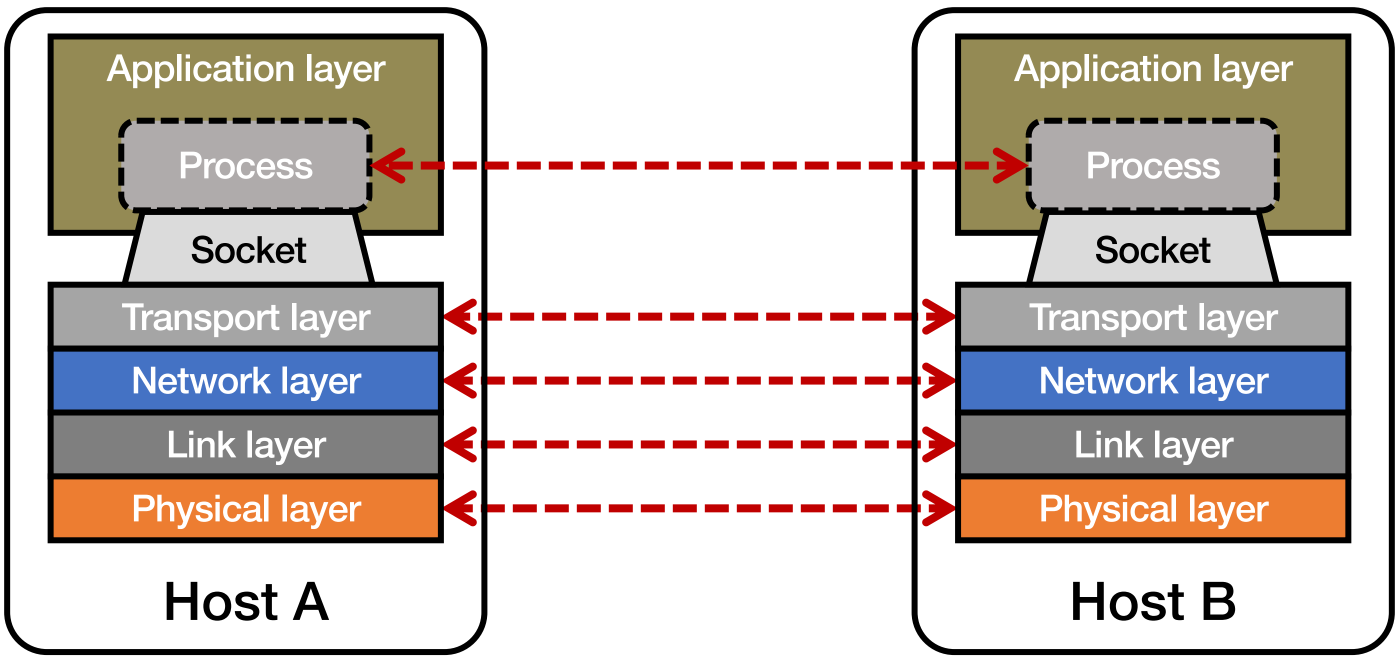
- How do peer protocols coordinate with each other?
- Layer attaches its own header (H) to communicate with peer
  - Higher layers' headers, data encapsulated inside message
    - Lower layers don't generally inspect higher layers' headers



Network socket-based communication

- **Socket:** The interface the OS provides to the network
  - Provides inter-process explicit message exchange

- Can build distributed systems atop sockets: `send()`, `recv()`
  - e.g.: **`put(key,value)`** → message



Network sockets: Summary

- Principle of transparency: Hide that resource is physically distributed across multiple computers
  - Access resource same way as locally
  - Users can't tell where resource is physically located

> Network sockets provide apps with point-to-point communication between processes

- **`put(key,value)`** → message with sockets?

```c
// Create a socket for the client
if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("Socket creation");
  exit(2);
}

// Set server address and port
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = inet_addr(argv[1]);
servaddr.sin_port = htons(SERV_PORT); // to big-endian

// Establish TCP connection
if (connect(sockfd, (struct sockaddr *) &servaddr,
            sizeof(servaddr)) < 0) {
  perror("Connect to server");
  exit(3);
}

// Transmit the data over the TCP connection
send(sockfd, buf, strlen(buf), 0);
```

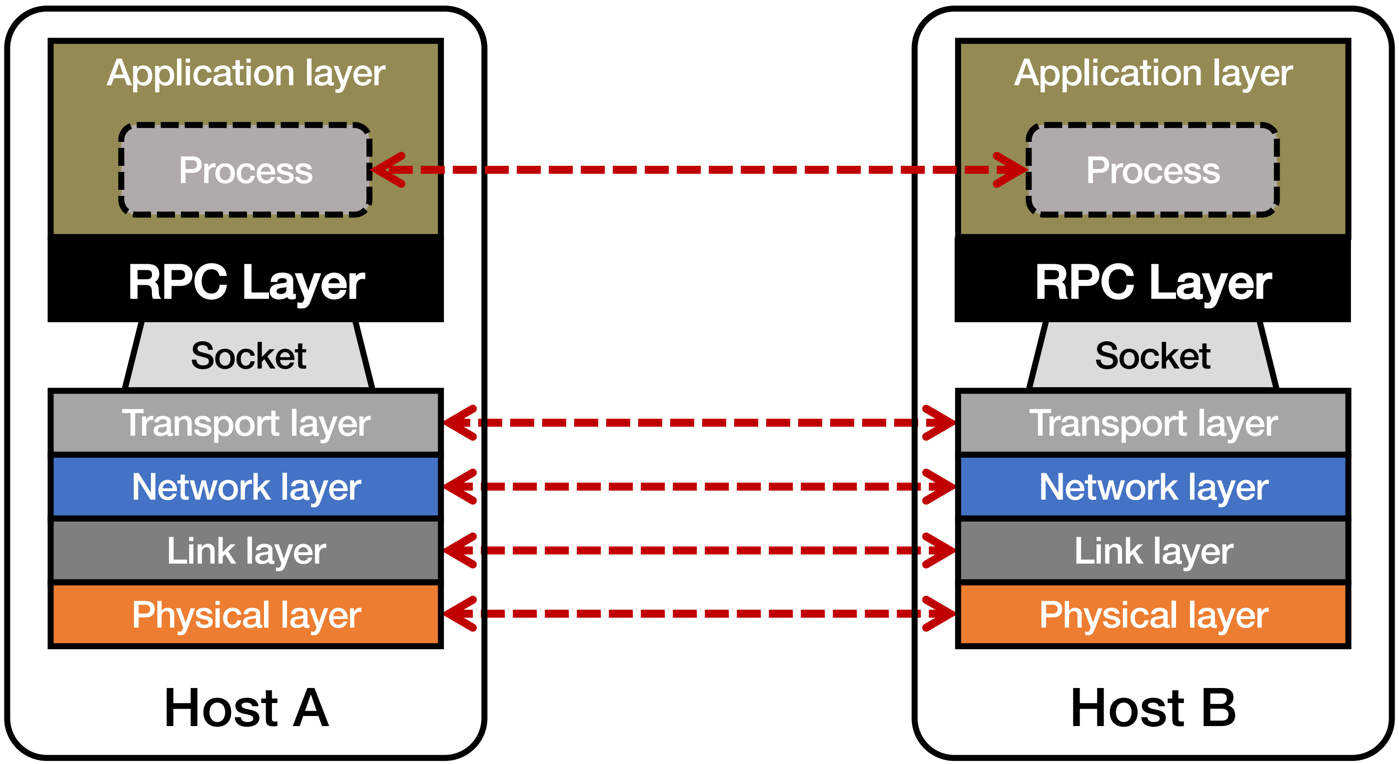
```c
// Create a socket for the client
if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("Socket creation");
  exit(2);
}

// Set server address and port
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = inet_addr(argv[1]);
servaddr.sin_port = htons(SERV_PORT); // to big-endian

// Establish TCP connection
if (connect(sockfd, (struct sockaddr *) &servaddr,
            sizeof(servaddr)) < 0) {
  perror("Connect to server");
  exit(3);
}

// Transmit the data over the TCP connection
send(sockfd, buf, strlen(buf), 0);
```

Sockets don’t provide transparency

Takeaway: Socket programming still not ideal (great)

- Lots for the programmer to deal with every time
    - How to separate different requests on the same connection?
    - How to write bytes to the network / read bytes from the network?
        - What if Host A's process is written in Go and Host B's process is in C++?
    - What to do with those bytes?
- Still pretty painful… Have to worry a lot about the network

Solution: Another layer!



Today's outline

How can processes on different cooperating computers exchange information?

1. Network sockets and raw messages

2. Remote procedure call

How can large computing jobs be parallelized?

3. MapReduce

Motivation: Why RPC?

- The typical programmer is trained to write single-threaded code that runs in one place
- **Goal:** Easy-to-program network communication that makes client-server communication transparent
    - Retains the “feel” of writing centralized code
        - Programmer needn’t think about the network
- Project 4-5 use Go RPC

What's the goal of RPC?

- Within a single program, running in a single process, recall the well-known notion of a procedure call:
  - Caller pushes arguments onto stack,
    - jumps to address of callee function
  - Callee reads arguments from stack,
    - executes, puts return value in register,
    - returns to next instruction in caller

What's the goal of RPC?

- Within a single program, running in a single process, recall the well-known notion of a procedure call:
  - Caller pushes arguments onto stack,
    - jumps to address of callee function
  - Callee reads arguments from stack,
    - executes, puts return value in register,
    - returns to next instruction in caller

RPC's Goal: make communication appear like a local procedure call: transparency for procedure calls – way less painful than sockets…

RPC issues

1. Heterogeneity
   - Client needs to rendezvous with the server
   - Server must dispatch to the required function
     - What if server is different type of machine?

RPC issues

1. Heterogeneity
   - Client needs to rendezvous with the server
   - Server must dispatch to the required function
     - What if server is different type of machine?

2. Failure
   - What if messages get dropped?
   - What if client, server, or network fails?

RPC issues

1. Heterogeneity
   - Client needs to rendezvous with the server
   - Server must dispatch to the required function
     - What if server is different type of machine?

2. Failure
   - What if messages get dropped?
   - What if client, server, or network fails?

3. Performance
   - Procedure call takes takes ≈ 10 cycles ≈ 3 ns
   - RPC in a data center takes ≈ 10 µs (10^3× slower)
     - In the wide area, typically 10^6× slower

RPC issues

1. Heterogeneity
   - Client needs to rendezvous with the server
   - Server must dispatch to the required function
     - What if server is different type of machine?

Today's lecture

2. Failure
   - What if messages get dropped?
   - What if client, server, or network fails?

3. Performance
   - Procedure call takes takes ≈ 10 cycles ≈ 3 ns
   - RPC in a data center takes ≈ 10 μs (10^3× slower)
     - In the wide area, typically 10^6× slower

Problem: Differences in data representation

- Not an issue for local procedure calls

- For a remote procedure call, a remote machine may:
  - Run process written in a different language
  - Represent data types using different sizes
  - Use a different byte ordering (endianness)
  - Represent floating point numbers differently
  - Have different data alignment requirements
    - e.g., 4-byte type begins only on 4-byte memory boundary

Problem: Differences in programming support

- Language support varies:
  - Many programming languages have no inbuilt way of extracting values from complex types
    - C, C++
    - Effectively need sockets glue code underneath
  - Some languages have support that enables RPC
    - Python, Go
    - Exploit type system for some help

Solution: Interface Description Language

- Mechanism to pass procedure parameters and return values in a machine-independent way

- Programmer may write an interface description in the IDL
  - Defines API for procedure calls: names, parameter/return types

Solution: Interface Description Language

- Mechanism to pass procedure parameters and return values in a machine-independent way

- Programmer may write an interface description in the IDL
  - Defines API for procedure calls: names, parameter/return types

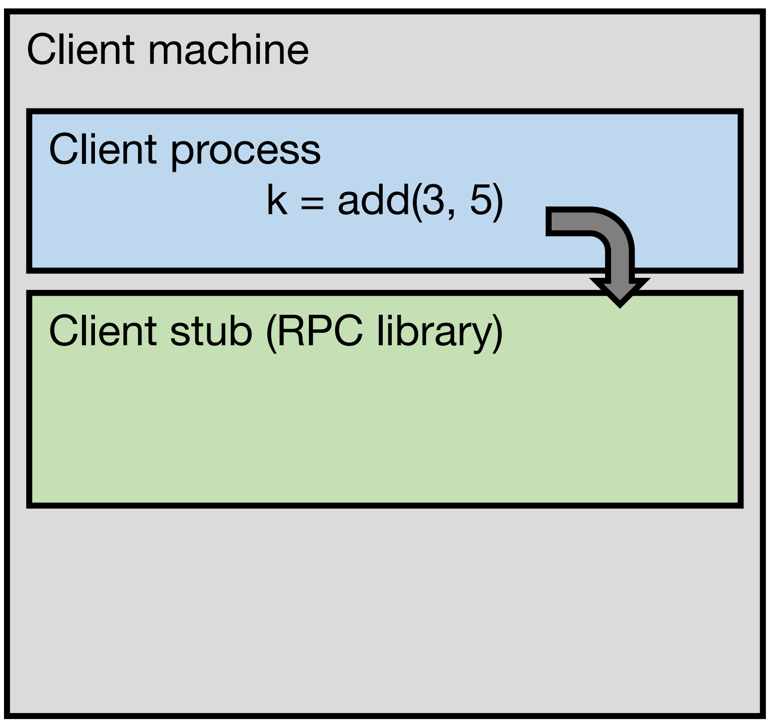
- Then runs an IDL compiler which generates:
  - Code to marshal (convert) native data types into machine-independent byte streams
    - And vice-versa, called unmarshaling

  - Client stub: Forwards local procedure call as a request to server

  - Server stub: Dispatches RPC to its implementation

A day in the life of an RPC

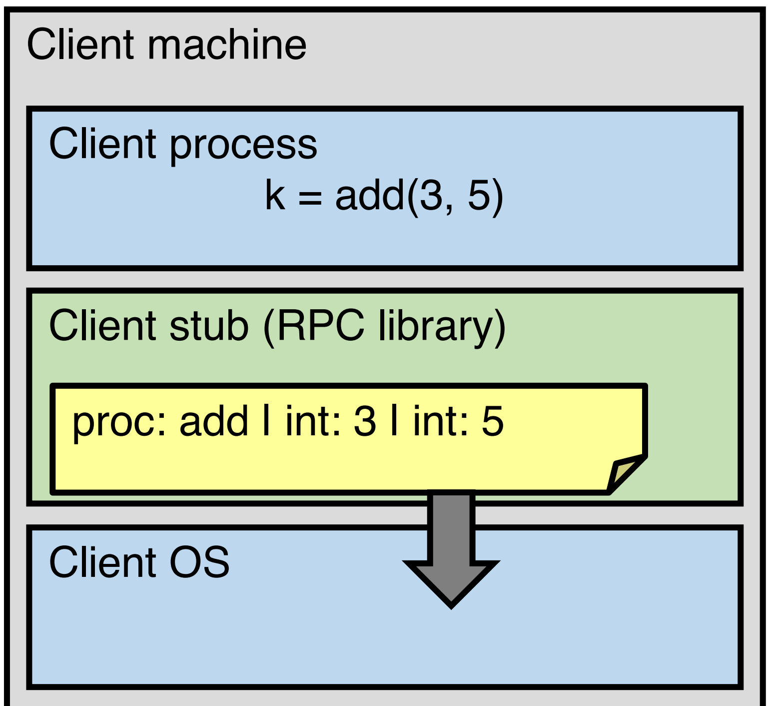
1. Client calls stub function (pushes parameters onto stack)



A day in the life of an RPC

1. Client calls stub function (pushes parameters onto stack)

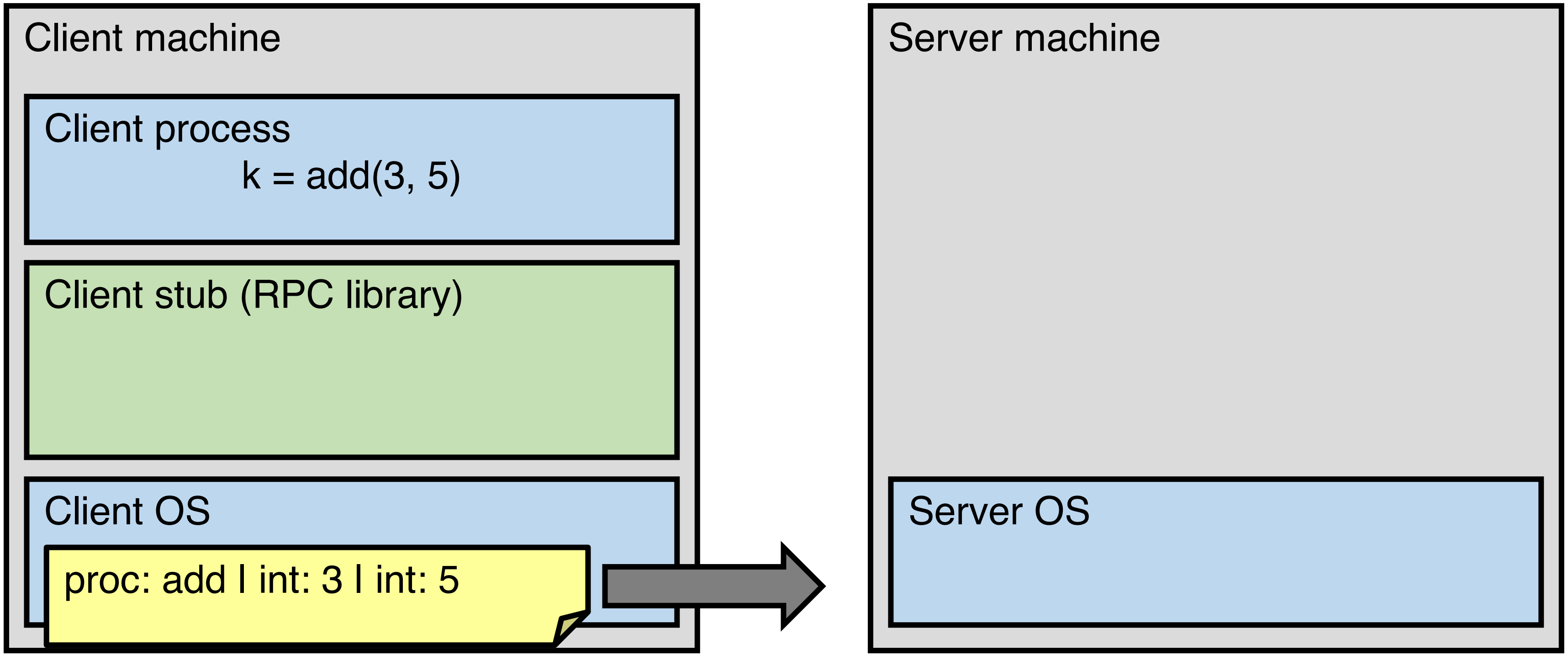
2. Stub marshals parameters to a network message



A day in the life of an RPC

2. Stub marshals parameters to a network message

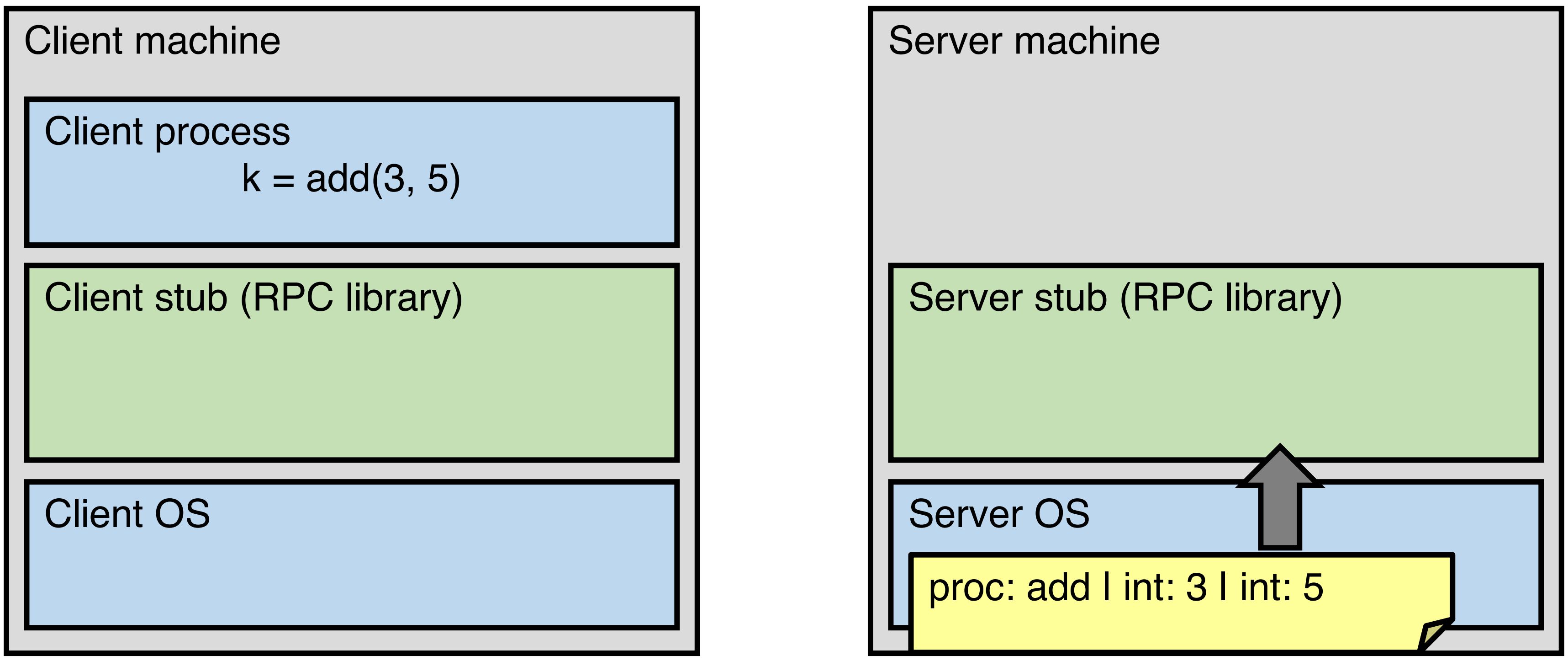
3. OS sends a network message to the server



A day in the life of an RPC

3. OS sends a network message to the server

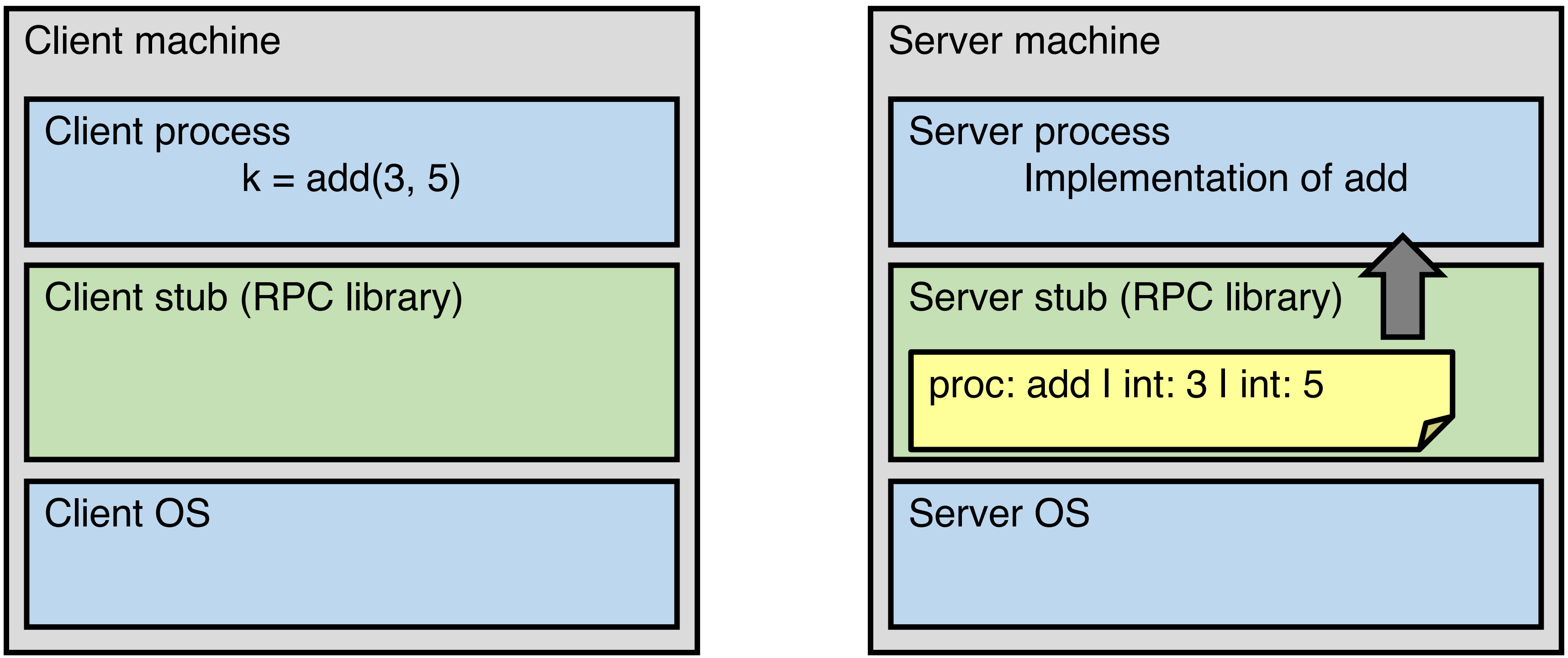
4. Server OS receives message, sends it up to stub



A day in the life of an RPC

4. Server OS receives message, sends it up to stub

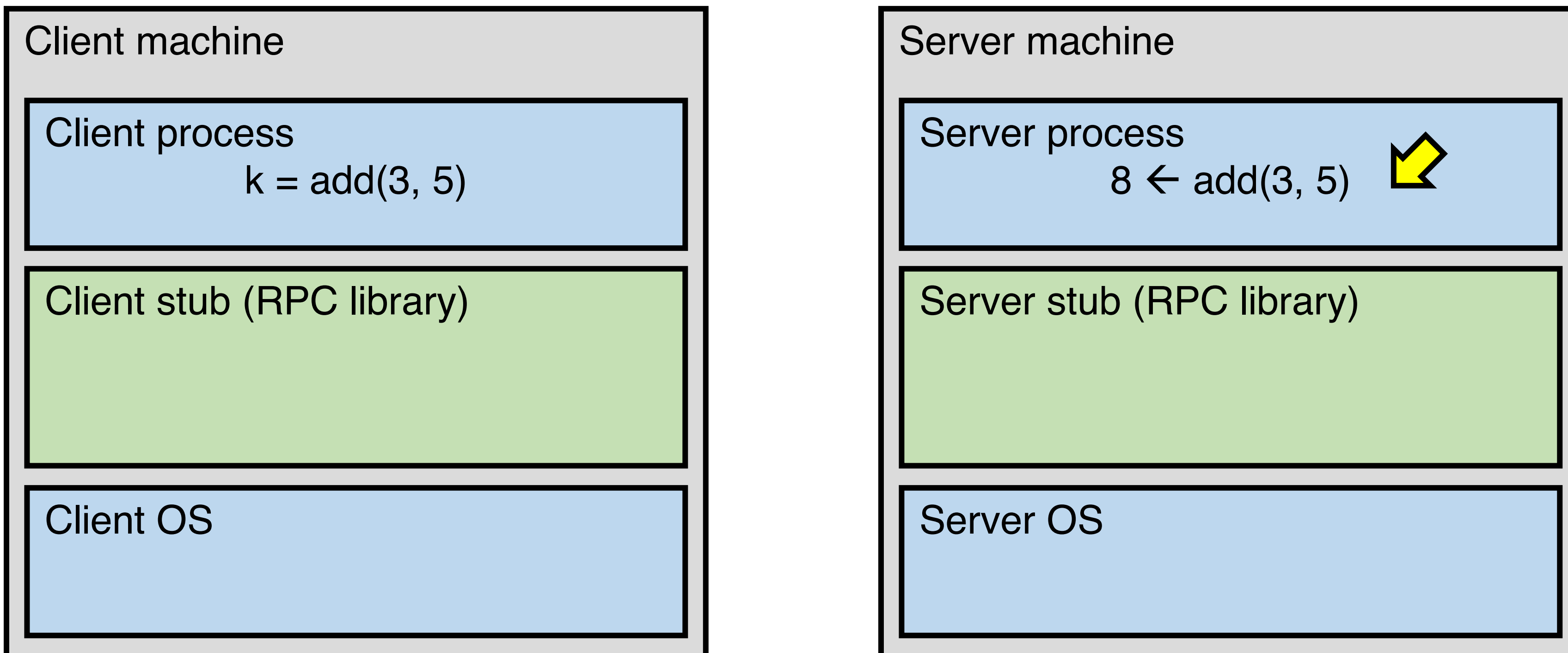
5. Server stub unmarshals params, calls server function



A day in the life of an RPC

5. Server stub unmarshals params, calls server function

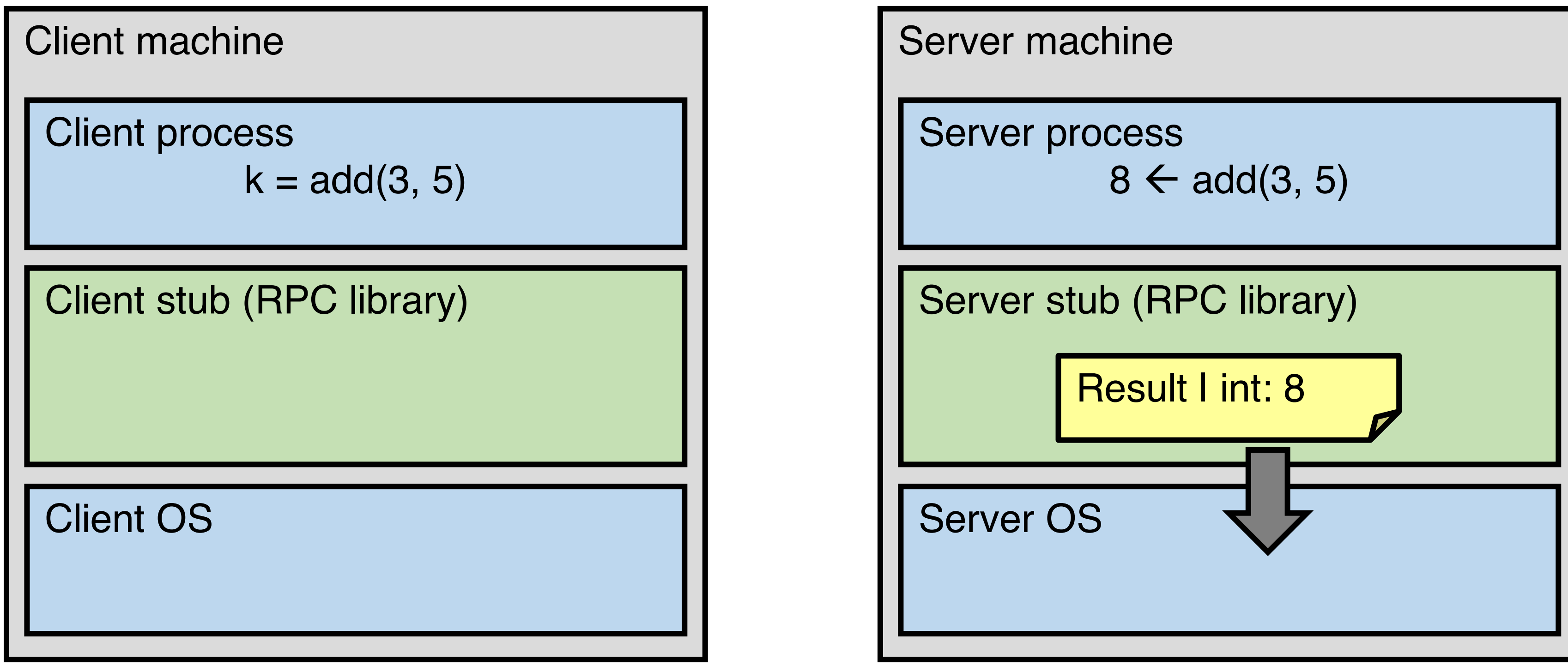
6. Server function runs, returns a value



A day in the life of an RPC

6. Server function runs, returns a value

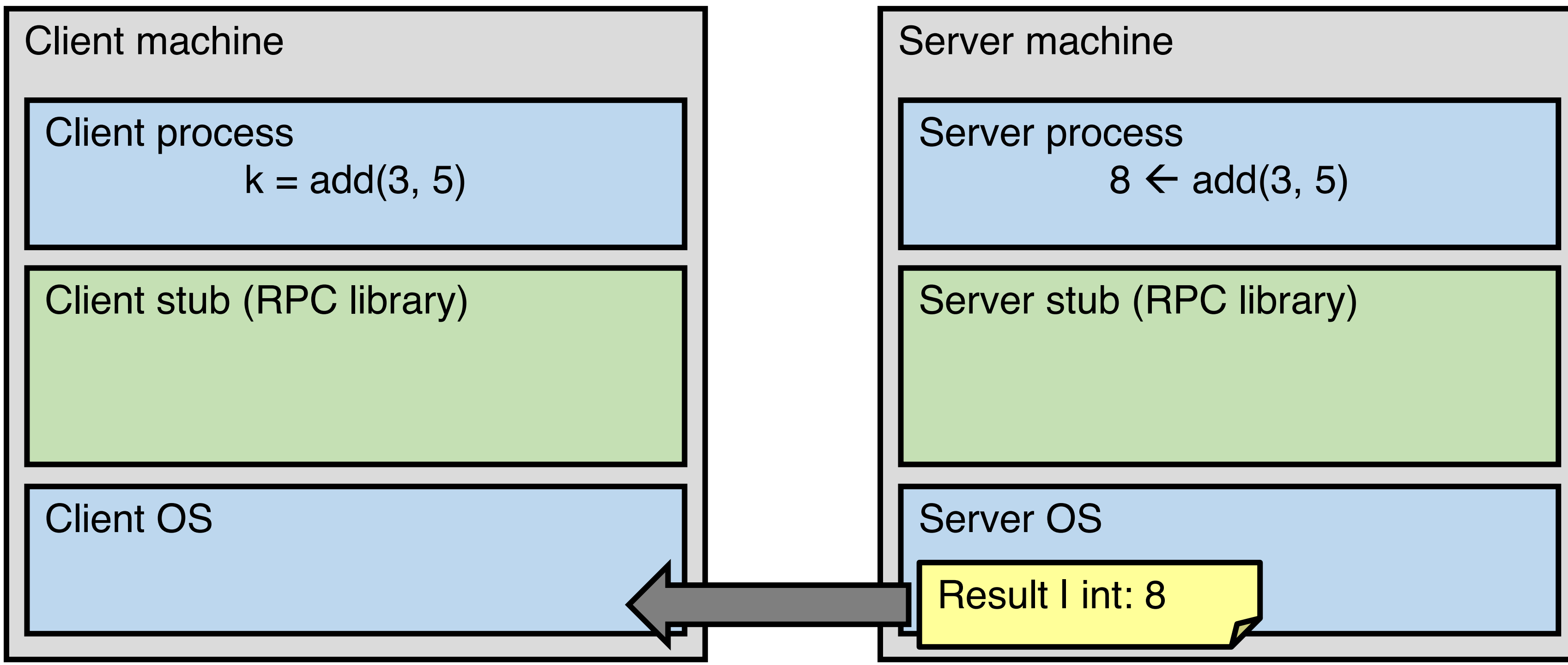
7. Server stub marshals the return value, sends message



A day in the life of an RPC

7. Server stub marshals the return value, sends message

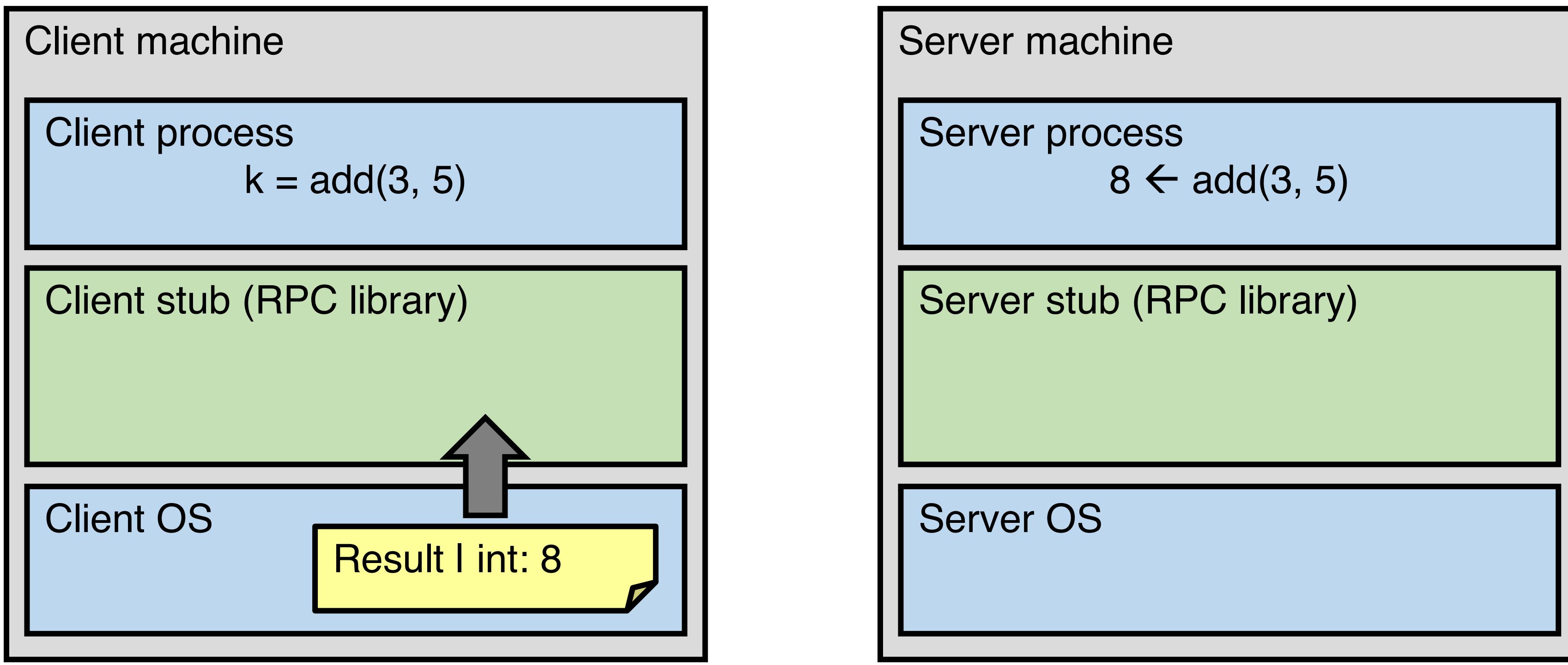
8. Server OS sends the reply back across the network



A day in the life of an RPC

8. Server OS sends the reply back across the network

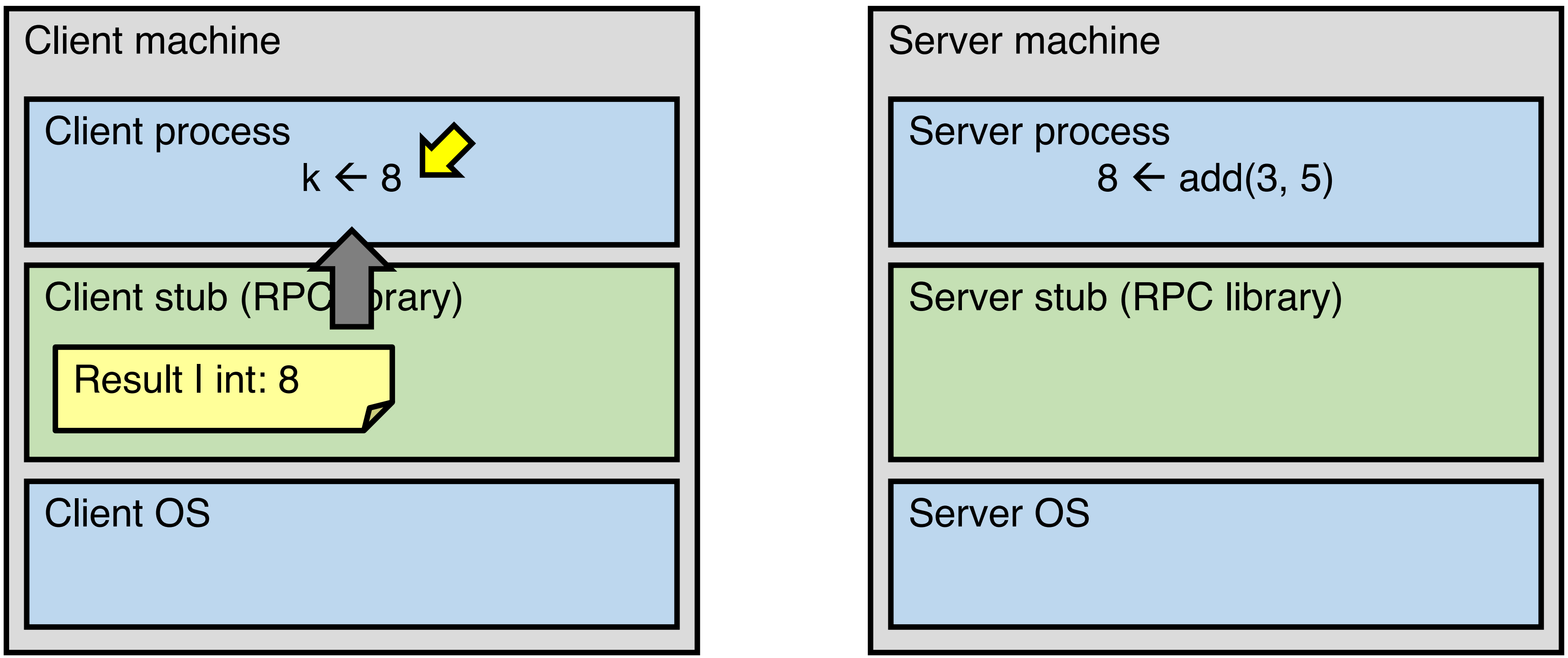
9. Client OS receives the reply and passes up to stub



A day in the life of an RPC

9. Client OS receives the reply and passes up to stub

10. Client stub unmarshals return value, returns to client



The server stub is really two parts

- Dispatcher
  - Receives a client's RPC request
    - Identifies appropriate server-side method to invoke
- Skeleton
  - Unmarshals parameters to server-native types
  - Calls the local server procedure
  - Marshals the response, sends it back to the dispatcher
- All this is hidden from the programmer
  - Dispatcher and skeleton may be integrated
    - Depends on implementation

Today's outline

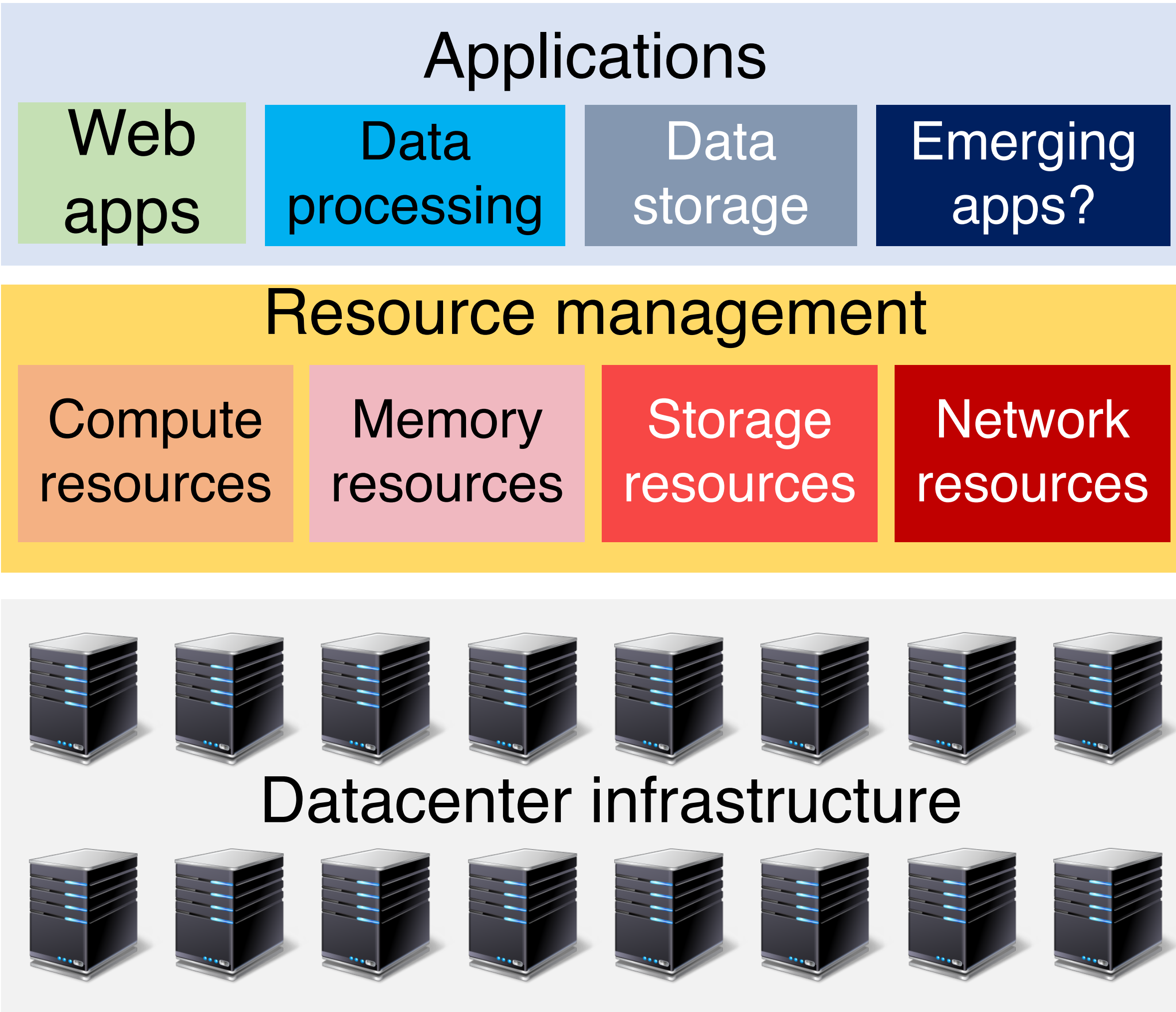
How can processes on different cooperating computers exchange information?

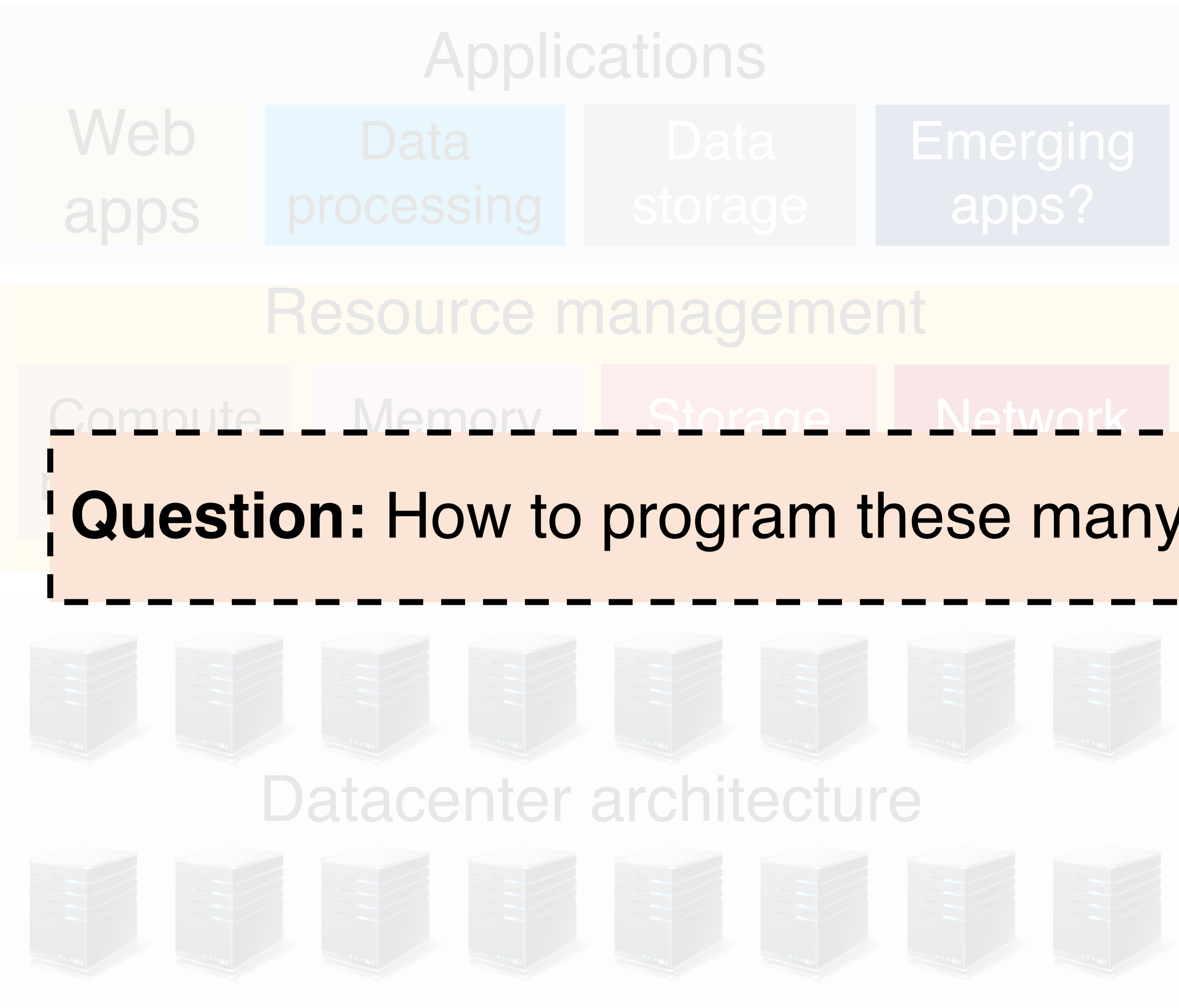
1. Network sockets and raw messages

2. Remote procedure call

How can large computing jobs be parallelized?

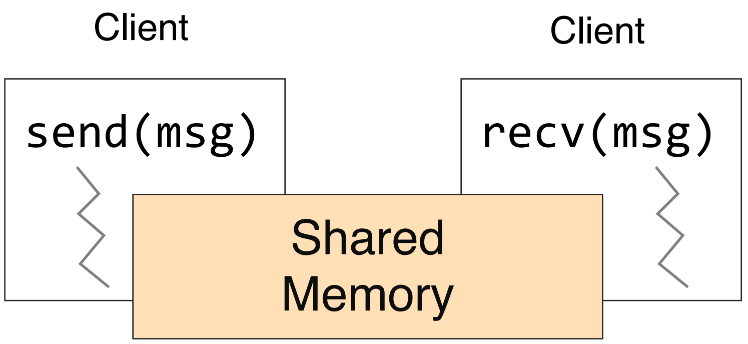
3. MapReduce


Applications
Web apps
Data processing
Data storage
Emerging apps?
Resource management
Compute resources
Memory resources
Storage resources
Network resources
Datacenter infrastructure



Question: How to program these many computers?

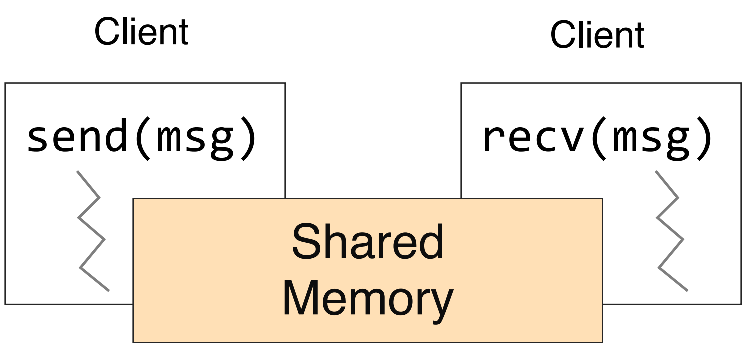
Shared memory



- Shared memory: multiple processes to share data via memory

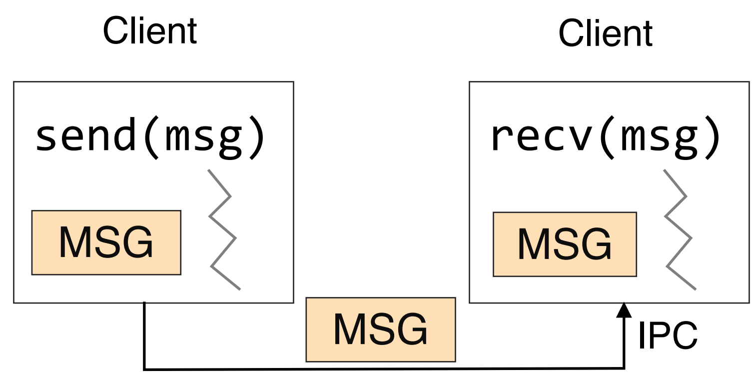
- Applications must locate and and map shared memory regions to exchange data

Shared memory vs. Message passing



- Shared memory: multiple processes to share data via memory

- Applications must locate and and map shared memory regions to exchange data



- Message passing: exchange data explicitly via IPC

- Application developers define protocol and exchanging format, number of participants, and each exchange

Shared memory vs. Message passing

- Easy to program; just like a single multi-threaded machines
- Hard to write high perf. apps:
  - Cannot control which data is local or remote (remote mem. access much slower)
- Hard to mask failures

- Message passing: can write very high perf. apps
- Hard to write apps:
  - Need to manually decompose the app, and move data
- Need to manually handle failures

Shared memory: Pthread

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- API specifies behavior of the thread library, implementation is up to development of the library

- Common in UNIX (e.g., Linux) OSes

Shared memory: Pthread

```
void *myThreadFun(void *vargp) {
      sleep(1);
      printf(“Hello world\n”);
      return NULL;
}

int main() {
      pthread_t thread_id_1, thread_id_2;
      pthread_create(&thread_id_1, NULL, myThreadFun, NULL);
      pthread_create(&thread_id_2, NULL, myThreadFun, NULL);
      pthread_join(thread_id_1, NULL);
      pthread_join(thread_id_2, NULL);
      exit(0);
}
```

Message passing: MPI

- MPI – Message Passing Interface
  - Library standard defined by a committee of vendors, implementers, and parallel programmers
  - Used to create parallel programs based on message passing

- Portable: one standard, many implementations
  - Available on almost all parallel machines in C and Fortran
  - De facto standard platform for the HPC community

Message passing: MPI

```
int main(int argc, char **argv) {
      MPI_Init(NULL, NULL);

      // Get the number of processes
      int world_size;
      MPI_Comm_size(MPI_COMM_WORLD, &world_size);

      // Get the rank of the process
      int world_rank;
      MPI_Comm_rank(MPI_COMM_WORLD, *world_rank);

      // Print off a hello world message
      printf(“Hello world from rank %d out of %d processors\n”,
            world_rank, workld_size);

      // Finalize the MPI environment
      MPI_Finalize();
}
```

Message passing: MPI

```
mpirun -n 4 -f host_file ./mpi_hello_world
```

```
int main(int argc, char **argv) {
      MPI_Init(NULL, NULL);

      // Get the number of processes
      int world_size;
      MPI_Comm_size(MPI_COMM_WORLD, &world_size);

      // Get the rank of the process
      int world_rank;
      MPI_Comm_rank(MPI_COMM_WORLD, *world_rank);

      // Print off a hello world message
      printf("Hello world from rank %d out of %d processors\n",
            world_rank, workld_size);

      // Finalize the MPI environment
      MPI_Finalize();
}
```

MapReduce

The big picture (motivation)

- Datasets are too big to process using a single computer

The big picture (motivation)

- Datasets are too big to process using a single computer
- Good parallel processing engines are rare (back then in the late 90s)

The big picture (motivation)

- Datasets are too big to process using a single computer
- Good parallel processing engines are rare (back then in the late 90s)
- Want a parallel processing framework that:
  - is **general** (works for many problems)
  - is **easy to use** (no locks, no need to explicitly handle communication, no race conditions)
  - can **automatically parallelize** tasks
  - can **automatically handle machine failures**

Context (Google circa 2000)

- Starting to deal with massive datasets
- But also addicted to cheap, unreliable hardware
  - Young company, expensive hardware not practical
- Only a few expert programmers can write distributed programs to process them
  - Scale so large jobs can complete before failures

Context (Google circa 2000)

- Starting to deal with massive datasets
- But also addicted to cheap, unreliable hardware
  - Young company, expensive hardware not practical
- Only a few expert programmers can write distributed programs to process them
  - Scale so large jobs can complete before failures
- **Key question:** how can every Google engineer be imbued with the ability to write parallel, scalable, distributed, fault-tolerant code?
- **Solution:** abstract out the redundant parts
- Restriction: relies on job semantics, so restricts which problems it works for

Application: Word Count

```
cat data.txt
     | tr –s ‘[[:punct:][:space:]]’ ‘\n’
     | sort | uniq -c
```

```
SELECT count(word), word FROM data
     GROUP BY word
```

Deal with multiple files?

1. Compute word counts from individual files

Deal with multiple files?

1. Compute word counts from individual files

2. Then merge intermediate output

Deal with multiple files?

1. Compute word counts from individual files

2. Then merge intermediate output

3. Compute word count on merged outputs

What if the data is too big to fit in one computer?

1. In parallel, send to worker:
   - Compute word counts from individual files
   - Collect results, wait until all finished

What if the data is too big to fit in one computer?

1. In parallel, send to worker:
   - Compute word counts from individual files
   - Collect results, wait until all finished

2. Then merge intermediate output

What if the data is too big to fit in one computer?

1. In parallel, send to worker:
   - Compute word counts from individual files
   - Collect results, wait until all finished

2. Then merge intermediate output

3. Compute word count on merged intermediates

MapReduce: Programming interface

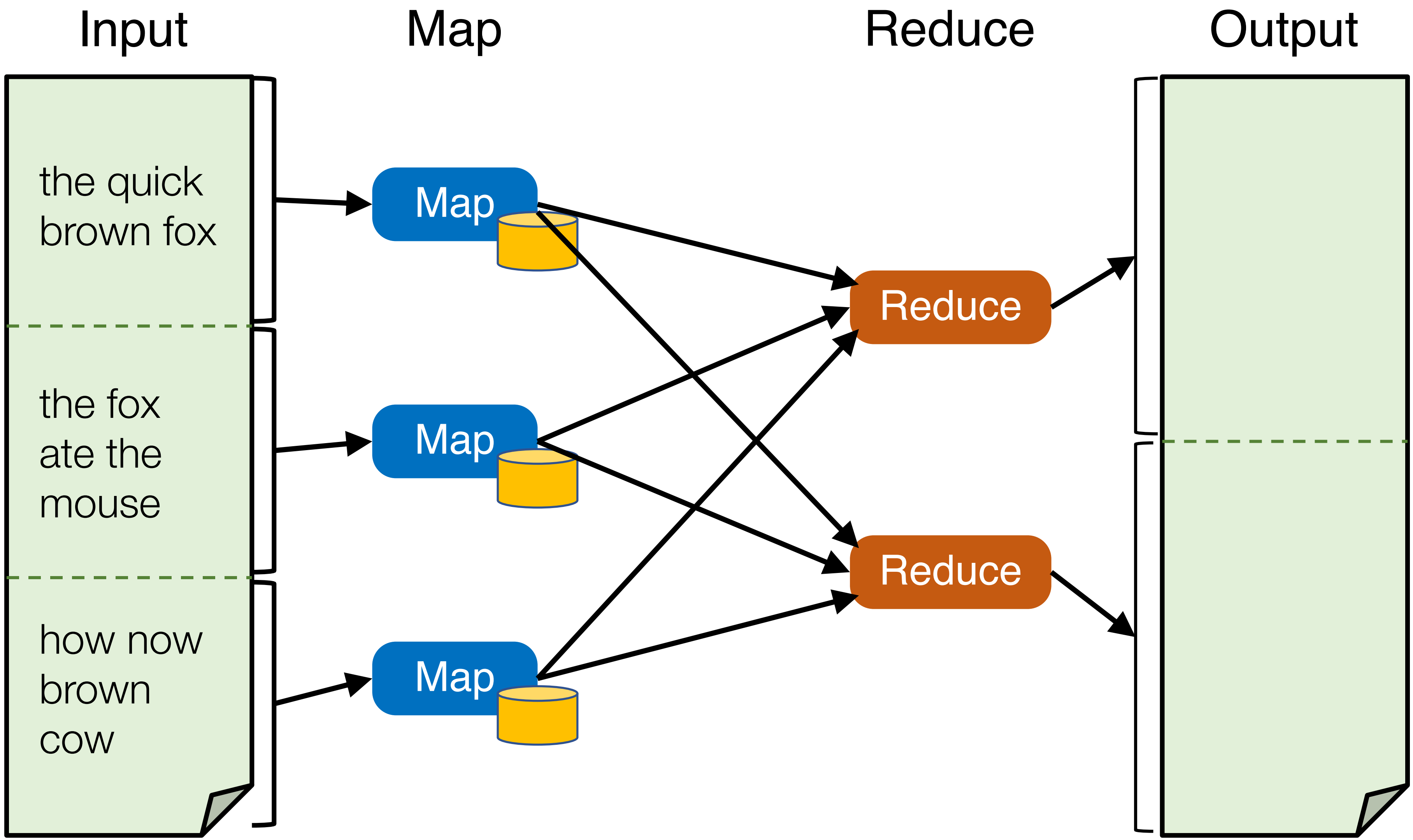
- `map(k1, v1)` → `list(k2, v2)`
  - Apply function to (`k1, v1`) pair and produce set of intermediate pairs (`k2, v2`)

- `reduce(k2, list(v2))` → `list(k3, v3)`
  - Apply aggregation (reduce) function to values
  - Output results

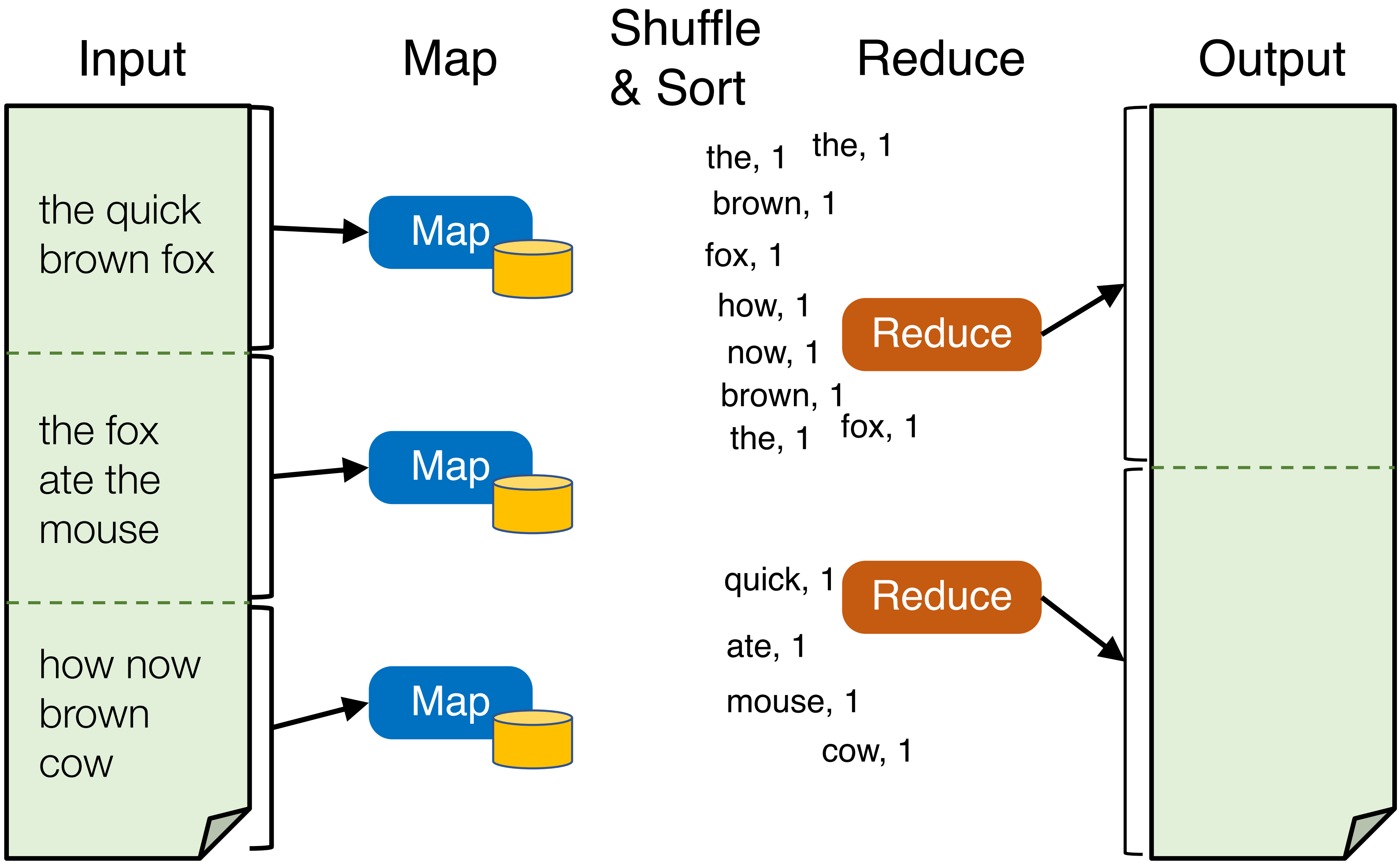
MapReduce: Word Count

```
map(key, value):
    for each word w in value:
        EmitIntermediate(w, “1”);

reduce(key, values):
    int result = 0;
    for each v in values:
        results += ParseInt(v);
    Emit(AsString(result));
```

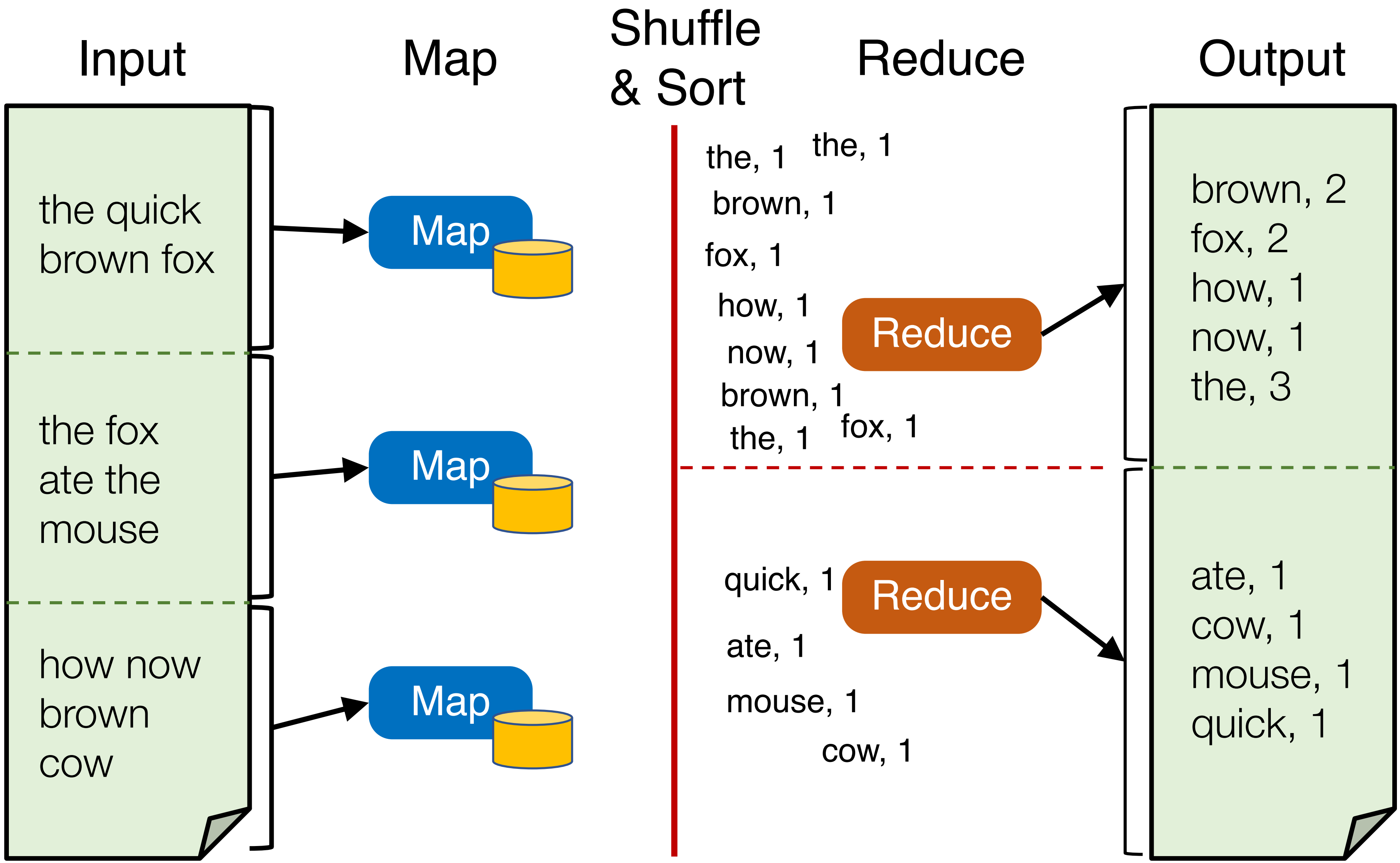
Word Count execution



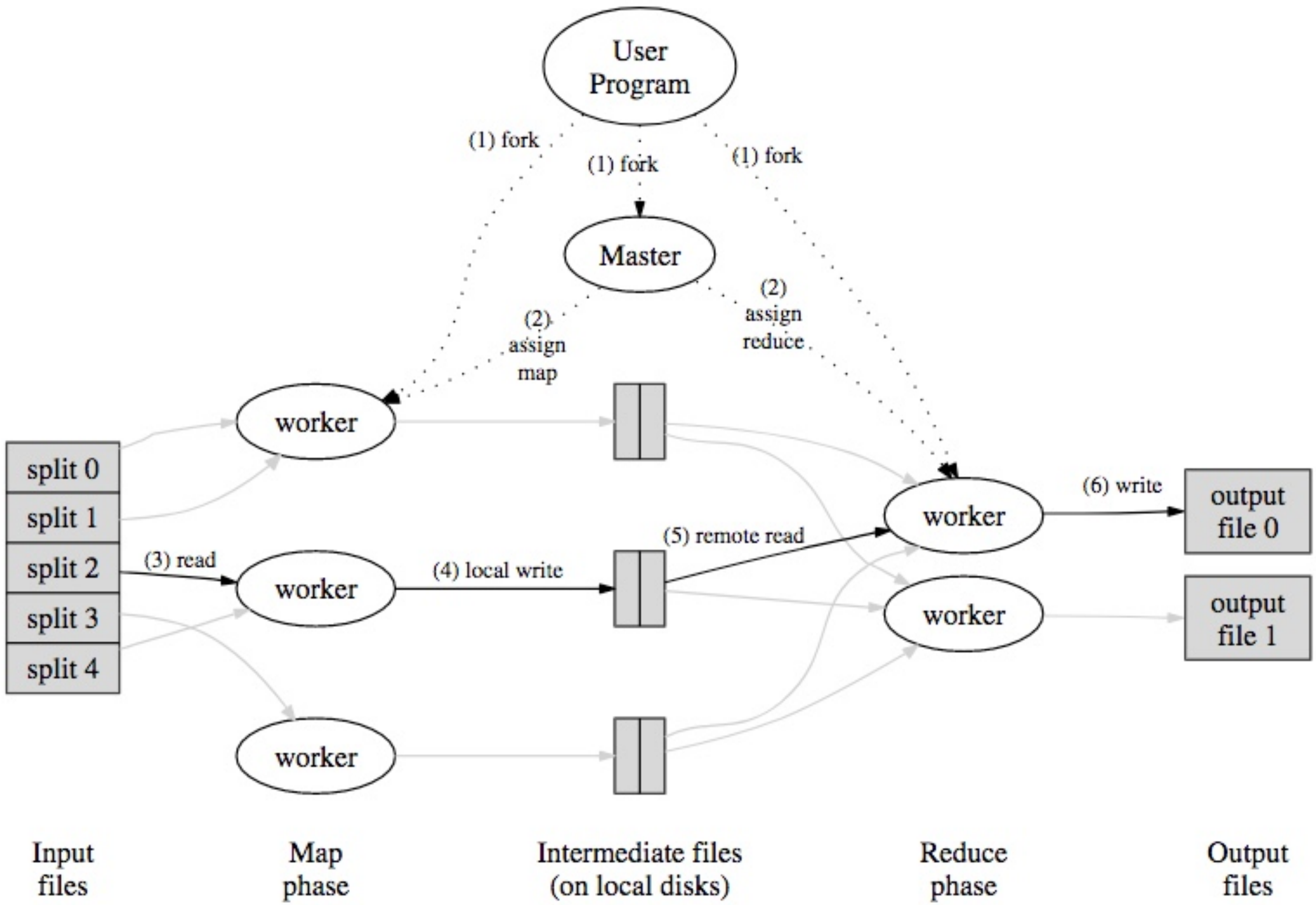
Word Count execution



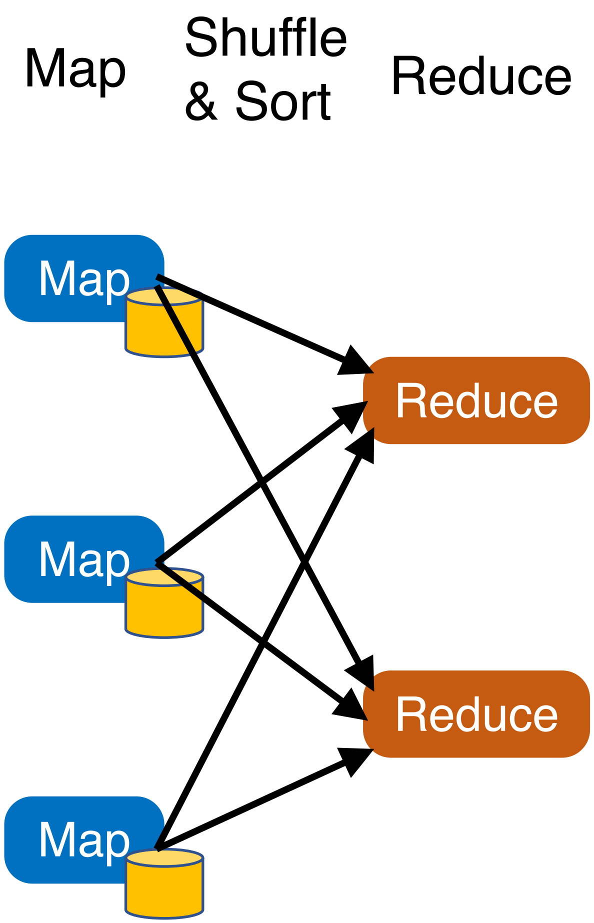
Word Count execution



MapReduce data flows



MapReduce processes



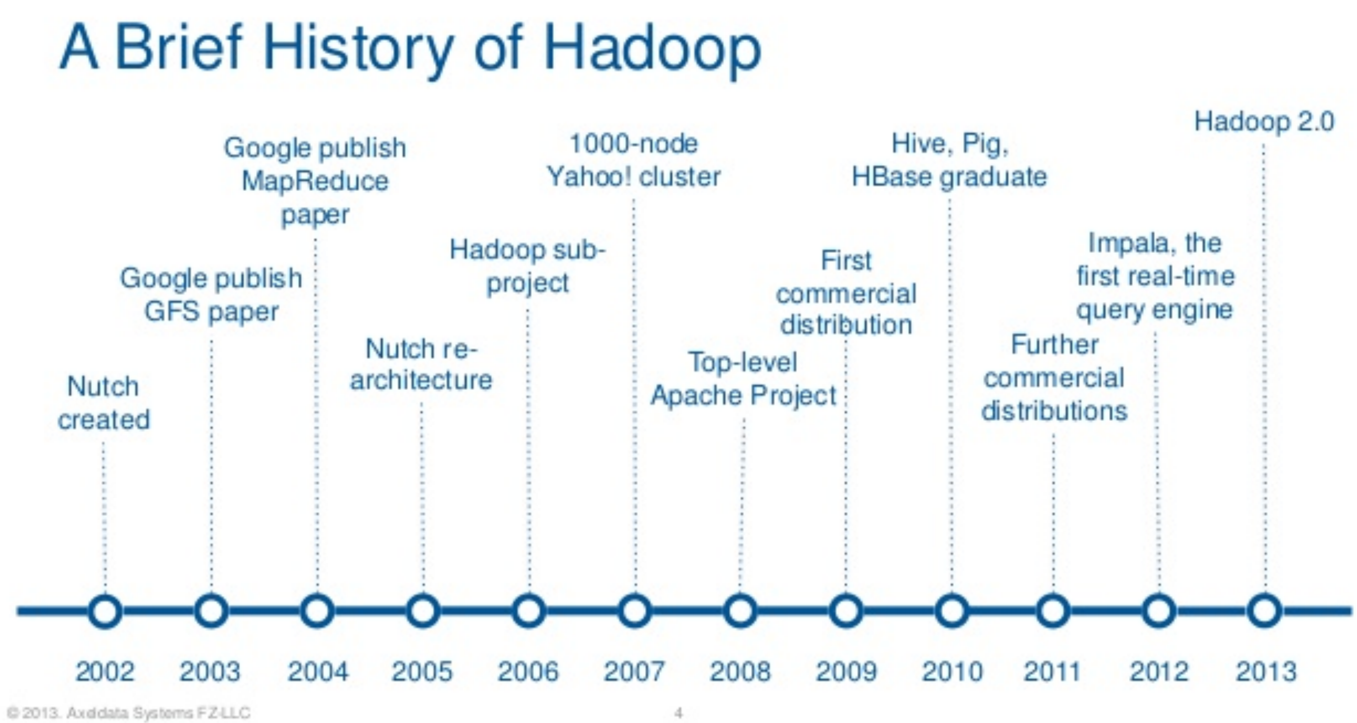
- Map workers write intermediate output to local disk, separated by partitioning. Once completed, tell master node

- Reduce worker told of location of map task outputs, pulls their partition's data from each mapper, execute function across data

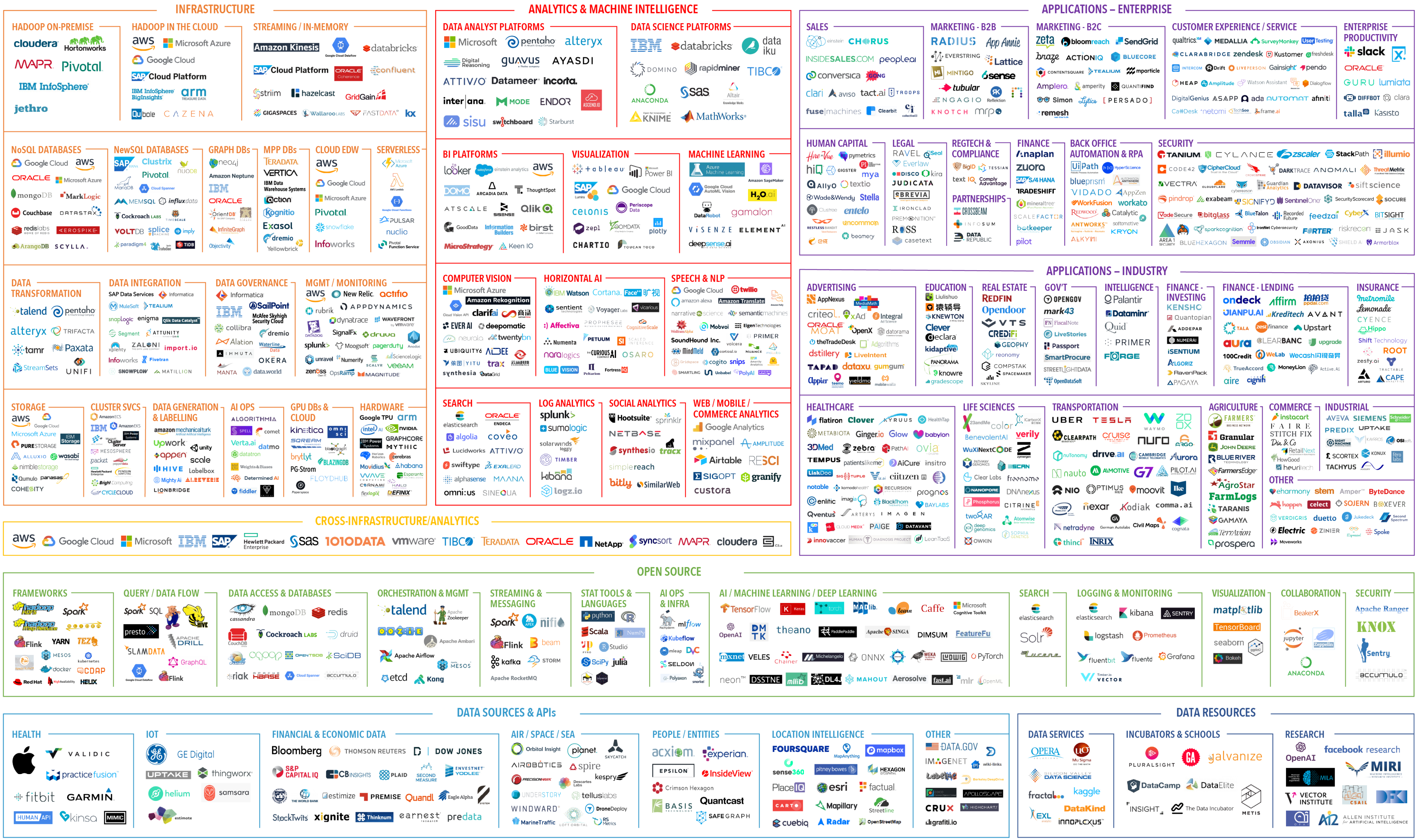
- Note:
  - "All-to-all" shuffle b/w mappers and reducers
  - Written to disk ("materialized") b/w each state

Apache Hadoop

- An open-source implementation of Google's MapReduce framework
  - Hadoop MapReduce atop Hadoop Distributed File System (HDFS)



DATA & AI LANDSCAPE 2019



July 16, 2019 - FINAL 2019 VERSION

© Matt Turck (@mattturck), Lisa Xu (@lisaxu92), & FirstMark (@firstmarkcap) mattturck.com/data2019

FIRSTMARK
EARLY STAGE VENTURE CAPITAL

Go RPC

Go RPCs

- Implementation in built-in library net/rpc

Go RPCs

- Implementation in built-in library net/rpc

- Write stub receiver methods of the form
  - `func (t *T) MethodName(args T1, reply *T2) error`

Go RPCs

- Implementation in built-in library net/rpc

- Write stub receiver methods of the form
  - `func (t *T) MethodName(args T1, reply *T2) error`

- Register receiver methods

Go RPCs

- Implementation in built-in library net/rpc
- Write stub receiver methods of the form
  - `func (t *T) MethodName(args T1, reply *T2) error`
- Register receiver methods
- Create a listener (i.e., server) that accepts requests

Writing a WordCount RPC server in Go

```
type WordCountServer struct {
	addr string
}

type WordCountRequest struct {
	Input string
}

type WordCountReply struct {
	Counts map[string]int
}
```

Writing a WordCount RPC server in Go

```
type WordCountServer struct {
      addr string
}

type WordCountRequest struct {
      Input string
}

type WordCountReply struct {
      Counts map[string]int
}
```

```
func (*WordCountServer) Compute(
            request WordCountRequest,
            reply *WordCountReply) error {
      counts := make(map[string]int)
      input := request.Input
      tokens := strings.Fields(input)
      for _, t := range tokens {
            counts[t] += 1
      }
      reply.Counts = counts
      return nil
}
```

Writing a WordCount RPC server in Go

```go
type WordCountServer struct {
      addr string
}

type WordCountRequest struct {
      Input string
}

type WordCountReply struct {
      Counts map[string]int
}
```

```go
func (*WordCountServer) Compute(
            request WordCountRequest,
            reply *WordCountReply) error {
      counts := make(map[string]int)
      input := request.Input
      tokens := strings.Fields(input)
      for _, t := range tokens {
            counts[t] += 1
      }
      reply.Counts = counts
      return nil
}
```

Writing a WordCount RPC server in Go

```
func (server *WordCountServer) listen() {
	rpc.Register(server)
	listener, err := net.Listen("tcp", server.addr)
	checkError(err)
	go func() {
		rpc.Accept(listener)
	}()
}
```

Writing a WordCount RPC server in Go

```
func (server *WordCountServer) listen() {
      rpc.Register(server)
      listener, err := net.Listen("tcp", server.addr)
      checkError(err)
      go func() {
            rpc.Accept(listener)
      }()
}
```

Writing a WordCount RPC server in Go

```
func (server *WordCountServer) listen() {
	rpc.Register(server)
	listener, err := net.Listen("tcp", server.addr)
	checkError(err)
	go func() {
		rpc.Accept(listener)
	}()
}
```

Writing a WordCount RPC server in Go

```
func (server *WordCountServer) listen() {
      rpc.Register(server)
      listener, err := net.Listen("tcp", server.addr)
      checkError(err)
      go func() {
            rpc.Accept(listener)
      }()
}
```

WordCount client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

WordCount client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {
      client, err := rpc.Dial("tcp", serverAddr)
      checkError(err)
      args := WordCountRequest{input}
      reply := WordCountReply{make(map[string]int)}
      err = client.Call("WordCountServer.Compute", args, &reply)
      if err != nil {
            return nil, err
      }
      return reply.Counts, nil
}
```

WordCount client

```go
func makeRequest(input string, serverAddr string) (map[string]int, error) {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

WordCount client

```
func makeRequest(input string, serverAddr string) (map[string]int, error) {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

WordCount client-server

```
func main() {
	serverAddr := "localhost:8888"
	server := WordCountServer{serverAddr}
	server.listen()
	input1 := "hello I am good hello bye bye bye bye good night hello"
	wordcount, err := makeRequest(input1, serverAddr)
	checkError(err)
	fmt.Printf("Result: %v\n", wordcount)
}
```

WordCount client-server

```
func main() {
	serverAddr := "localhost:8888"
	server := WordCountServer{serverAddr}
	server.listen()
	input1 := "hello I am good hello bye bye bye bye good night hello"
	wordcount, err := makeRequest(input1, serverAddr)
	checkError(err)
	fmt.Printf("Result: %v\n", wordcount)
}
```

```
Result: map[hello:3 I:1 am:1 good:2 bye:4 night:1]
```

Is this synchronous or asynchronous?

```
func makeRequest(input string, serverAddr string) (map[string]int, error)
{
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	err = client.Call("WordCountServer.Compute", args, &reply)
	if err != nil {
		return nil, err
	}
	return reply.Counts, nil
}
```

Making client asynchronous

```
func makeRequest(input string, serverAddr string) chan Result {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}









	return ch
}
```

Making client asynchronous

```
func makeRequest(input string, serverAddr string) chan Result {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	ch := make(chan Result)
	go func() {
		err := client.Call("WordCountServer.Compute", args, &reply)
		if err != nil {
			ch <- Result{nil, err} // something went wrong
		} else {
			ch <- Result{reply.Counts, nil} // success
		}
	}()
	return ch
}
```

Making client asynchronous

```
func makeRequest(input string, serverAddr string) *rpc.Call {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	return client.Go("WordCountServer.Compute", args, &reply, nil)
}
```

Making client asynchronous

```go
func makeRequest(input string, serverAddr string) *rpc.Call {
	client, err := rpc.Dial("tcp", serverAddr)
	checkError(err)
	args := WordCountRequest{input}
	reply := WordCountReply{make(map[string]int)}
	return client.Go("WordCountServer.Compute", args, &reply, nil)
}
```

```go
call := makeRequest(...)
<-call.Done
checkError(call.Error)
handleReply(call.Reply)
```

Next lecture

- Google File System (GFS) and Network File System (NFS)

- Read the GFS paper